Constraint programming for user-interface construction

Samuel, John

711

Constraint programming for user-interface construction

John Samuel

Submitted for the degree of Doctor of Philosophy
Queen Mary and Westfield College
University of London

Constraint programming for user-interface construction

John Samuel

Submitted for the degree of Doctor of Philosophy
Queen Mary and Westfield College
University of London
December 1995

Abstract

Constraint programming and techniques of constraint satisfaction are areas of active research in several disciplines of Computer Science: logic programming, artificial intelligence and human-computer interaction. In the field of human-computer interaction, constraint programming can simplify the task of user interface construction by providing the designer with a declarative means to specify the implementation of user interfaces and user-interface toolkits.

From the point of view of user interface design, the domain of the constraints and the techniques employed to satisfy them have implications for the usability of the resulting system. Previous research on using constraints in user interface construction has concentrated on a simple form of constraint satisfaction, known as local propagation, which has the advantage of efficient implementation, but at the cost of limited expressiveness and effectiveness. A particular problem with these systems is that they do not have effective abstractions for describing the relationship between graphical objects in the interface and the underlying functionality which they represent: even quite simple spatial relationships may be awkward or impossible to describe.

This thesis describes the ways in which the underlying constraint model influences the nature of the user interface under construction and examines alternatives to local propagation as a technique of constraint satisfaction. It presents a constraint system, based on results of recent work in logic programming and artificial intelligence, whose domain is specialized to describe the pictorial semantics of the user interface.

Acknowledgements

I would like to acknowledge the help of several colleagues who have been directly involved with the development of this thesis[1]: Peter Johnson who supervised it; George Colouris and Eliot Miranda who were on my PhD committee; and last, but not least, Panos Markopoulos and Stephanie Wilson for their patient help in 'debugging' the final draft.

Many other people have contributed to the research in an indirect way: through informal discussions and practical help and support. I include all members, past and present, of the HCI group at Queen Mary & Westfield College and members of the academic and support staff in the Computer Department of that College.

My thanks go to all of these people, without whose support, patience and good humour the inordinately long time which this thesis took to complete would have seemed an eternity.

1. The research for this thesis was supported by a grant from the Joint Council Initiative in Cognitive Science/HCI. SERC Award no 90312256.

Table of contents

List of figures

List of tables

The first gloss

Take hold of the shaft of the pen.
Subscribe to the first step taken
from a justified line
into the margin.

Seamus Heaney
From *Station Island, Part III, Sweeney Redivivus.*
Pub. Faber & Faber 1984

Chapter 1 Introduction

This chapter presents the background to the thesis and gives an outline of subsequent chapters. A substantial part of the chapter, section 1.1, concerns visual representations. This section describes, in broad terms, a theme that runs throughout the thesis: that graphical user interfaces may usefully be considered as examples of visual representations. We describe some of the problems of formalizing visual representations and indicate how these relate to the specific problem considered in the thesis, which is to give a formal account of how constraint programming may be used in the construction of user interfaces. In section 1.2 we give a brief overview of the position which is argued in the thesis. Finally, in section 1.3 we present a chapter-by-chapter summary of the thesis and describe how each chapter contributes to the development of the argument.

1.1 Visual representations

In this thesis, we consider graphical user interfaces as visual representations of information. The range of visual representations is large and they have been the subject of investigation for centuries. Tufte presents two good general reviews [150,151]. Recent years have seen a proliferation of new media for the representation and manipulation of visual information, many based on computer systems. Whereas much research has been carried out on ways to formalize the description of systems based on written language, little work exists to give a formal grounding for visual representations.

We will describe one particular class of system for visual representation of information, the *graphical user interface* and attempt a thorough exposition within that narrow setting. Much of the argument presented in the rest of this thesis will focus on technical concerns, related to such matters as system architecture, programming language design and solution of systems of constraints based on spatial relations. The common strand that links these various themes is the idea that in the user interface we are dealing with a visual representation and, therefore, that the narrowly defined issues of user-interface construction should be considered within the wider context of systems for visual representation.

In this thesis we will be concerned exclusively with representations in which information is encoded using spatial layouts. There are, of course, many other graphical properties which can be used to convey information, such as colour, texture and shape. The focus on spatial layouts should not be taken to imply that these other properties are less important. To give a complete account of an area as rich and complex as visual representations would be an enormous endeavour, so we have chosen to concentrate on this one aspect.

The detailed argument presented in this thesis is concerned with the application of a particular set of programming techniques to the specification and implementation of graphical user interfaces[1]. We will, for the moment, term the whole set of these techniques *constraint programming*. The use of this term is slightly loose, since strictly speaking, many of the techniques to which we refer are actually *constraint satisfaction* techniques. The former term implies the use of a programming language in which constraints are explicit in the syntax, the latter refers to algorithmic techniques for the solution of systems of constraints, independent of any specific programming language. In Chapter 3 we give a more detailed account of constraint programming and constraint satisfaction.

The idea of applying constraint programming to graphical interfaces is by no means a new one: it dates back to a system called Sketchpad, which was described by Sutherland in 1963 [143]. This used constraint programming within a design and drawing package. In spite of this long history (relative to the time scale of computer science), building highly graphical user interfaces using constraints remains a demanding and awkward task. In this thesis we will argue that the task may be simplified by extending the range of constraint-programming techniques which are available to the interface developer and by providing a set of abstractions, based on constraints, for describing visual aspects of the interface. We will argue that existing applications of constraint programming to the construction of user interfaces complicate the task of the designer by enforcing descriptions at an inappropriate level of detail. Furthermore, we will argue that, given an appropriate formulation of the constraint system, constraint programming may be given a central role in the description of user interface behaviour, rather than being confined to tools for the

1. We will use the term 'graphical user interface' when referring to concrete implementations of visual representations in computer-based applications and 'visual representation' when referring to representations in the abstract.

description of screen layout. To support this claim, we will present a user interface architecture, based entirely on constraint programming.

Taken narrowly, the thesis gives a formal description of a set of constraints with which we may define spatial properties of visual representations and of a notation, based on these constraints, for specifying graphical user interfaces. The declarative nature of constraint programming is one of its most appealing features as a tool for user interface specification and construction. However, it is well-known that one of the problems with constraint programming is the possibility it affords of specifying insoluble problems and, therefore, the consideration of efficient techniques for the solution of the constraints is an important secondary theme in this thesis.

The interface and its components are examples of visual representations. The problem of how to give meaning to visual representations has often been compared with the problem of defining semantics for languages [14]. A representation of any kind will have the following elements

- a represented world and a representing world;
- the modelled aspects of the represented world and the modelling aspects of the representing world;
- the correspondence between the two worlds.

In the systems with which we shall be concerned, the representing world will be a visual representation in 2.5D space, modelled by a set of graphical objects in a computer system. Spatial relations between the objects model aspects of the represented world. The represented world will be modelled by objects in an application program. For the present we will consider the representing and represented worlds as abstract entities separate from the underlying architectures which support them. We discuss user-interface architectures in detail in the next chapter.

We will argue that the use of an appropriate choice of constraint system to model the spatial relations in the interface greatly simplifies the task of defining the correspondence between the represented and representing worlds. The converse is easily demonstrated: given an inappropriate choice of constraint system, it is *practically impossible* to specify the correspondence. The reason for this is the huge amount of redundancy contained in many forms of visual representation. In many cases, the absolute location of objects contains no significant information whatsoever: it is only the spatial relations *between* objects that has any significance. For example, we might wish to say that when one object

contained another in the representation, then some predicate was true of the corresponding objects in the represented world. Given a description of the representing world in terms of the absolute coordinates of individual objects, even a relationship as simple as containment has numerous possible cases, each of which corresponds to the same state of the represented world. For this reason, most approaches to specification of user interfaces do not attempt to define explicitly the mapping between the state of the interface (the representing world) and the state of the underlying domain model (the represented world), rather they rely on context-dependent interpretation of user inputs. Such approaches have their own drawbacks, notably that, because the mapping is implicit, it is very hard to design systems which do not have a high degree of coupling between interface and the functional core (the code which implements the domain model). These issues are the subject of the next chapter when we examine user-interface architectures in more detail. For the moment, we merely observe that the nature and complexity of the mapping between the two worlds is greatly influenced by the choice of representation. Moreover, in order to achieve a simple correspondence between interface and domain model, we need a means to characterize the state of the interface which can cope adequately with the redundancy that is typically present.

In order to lay the foundations for a consistent and well-motivated approach to these questions, the rest of this section reviews relevant research into visual representations. We start out in section 1.1.1 by considering how to characterize forms of representation. We refer to research which suggests that certain forms of representation are easier to use because they better support inference. Section 1.1.2 extends this line of argument by considering visual representations as inferential systems. In section 1.1.3 we compare visual representations with text-based language. The user interface differs from diagrammatic forms of visual representation because of its dynamic character. In section 1.1.4 we consider how this affects the characterization of the user interface as a form of visual representation. Finally, we discuss briefly some of the cognitive aspects of visual representations.

1.1.1 Characterization of representations

It is the aim of this thesis to show that the correspondence between represented and representing worlds may be manageably simple when the user interface is described as a system of spatial constraints. We now consider how we may characterize that correspondence.

Stenning & Oberlander [138] suggest that the nature of the correspondence between represented and representing worlds has some bearing on the usability of the system. Many styles of interaction are derived from physical analogues, such as dials, sliders and gauges. It is often argued that the advantage of this is that users' familiarity with the analogues reduces the difficulty of learning the interface. However, it may be that the users' ease of understanding derives more from the nature of the representation of the components, rather than any previous experience with their physical counterparts. That is to say, the accessibility of such components may not be due to the existence of familiar analogues, but to some property possessed by both the analogue and the interface component.

Stenning & Oberlander consider the classification of visual representations and put forward three categories of representation, which may be distinguished formally in terms of a model-theoretic semantics. Objects such as sliders which determine unambiguously the values they represent fall into the category which they term *Minimal Abstract Representation Systems* (MARS). In model-theoretic terms, there is exactly one model for every representation in a MARS. This contrasts with other forms of representational system in which a given representation may correspond with more than one model. A MARS cannot represent indeterminacy in the value chosen: for example, a conventional slider does not allow the user to express uncertainty about the chosen value, say that it is restricted to lie within a given range. *Limited Abstract Representation Systems* (LARS) allow the expression of disjunction by explicit enumeration of cases, whereas *Universal Abstract Representation Systems* (UARS) place no restriction on disjunctions. The distinction between LARS and UARS is one of degree, rather than of kind.

The most general representation systems, such as first-order logic, are those in which a given representation may have many models. These are preferable in terms of their expressiveness, but their implementations are less efficient. As Stenning and Oberlander put it:

> By showing that the logic of graphical representations is more computationally tractable than more general logics, we show that processing the information graphical representations convey is easier than processing some more general class of information. This explains why graphical representations are easier to process *for any reasoning system.*
>
> ([138] Italics in original.)

The significance of the italicised phrase is that users are also 'reasoning systems' so that graphical representations may facilitate reasoning by users.

The concept of a MARS corresponds to Levesque's conception of *vivid knowledge*. This is characterized by a ' ... one-to-one correspondence between entities and relationships of interest in the world and certain symbols and appropriate connections between them in the vivid knowledge base' [95]. Knowledge that one of a set facts is true, without specifying which one, is *incomplete* in Levesque's terminology. The use only of vivid facts ensures the completeness of knowledge. Levesque defines a set of conditions for a knowledge-base to be vivid [96,155], *viz*

- function-free, ground sentences;
- unique names for entities and axioms of equality;
- closed-world assumption, i.e. every predicate is either explicitly true of a given set of constants, or assumed false.

Barwise and Etchemendy [7] define a similar concept to that of the MARS which they term *homomorphic* representations. They give a list of features that may be possessed by homomorphic representations, including the following.

- Representing objects correspond in a one-to-one way with represented objects.
- There is a mapping between types of representational objects and properties of represented objects.
- That mapping is structure-preserving, so that related types of representational objects correspond to represented objects with related properties. Higher-order properties of relations among the represented objects such as transitivity and symmetry should apply to the corresponding relations among the represented objects.
- The system should be complete and correct. i.e. every possible situation can be represented and every representation corresponds to a realizable situation in the represented world.

The concept of homomorphism in representations corresponds to a property which Green has described as *role-expressiveness* in the context of describing notations [61].

There is a close parallel between the way in which many visual representations encode information and the way in which vivid systems do so. The 'sentences' of a visual representation are structures built up from primitive symbols, in which information is encoded as properties of, and relations between,

the diagrammatic objects. They may be considered 'function-free and ground' whenever each symbol represents a unique, fully determined object in the represented world. The objects in a visual representation may be considered to be uniquely named whenever there is exactly one occurrence of the symbol which represents any given object. The closed world assumption implies that what is not shown in the representation does not exist in the represented world.

1.1.2 Visual representations as inferential systems

Casner observes that graphics allow users to make certain inferences directly from perception, rather than through logical reasoning [19]. Inferences based on size and position are often much more efficient than their counterparts based on number. Similarly, visual cues may improve the efficiency of search for particular classes of information, as when closely associated information is displayed in the same locality, or using a common encoding, say in colour. Casner's view is that any visual representation should be considered in conjunction with the particular set of tasks which it is designed to support. In this way, the perceptual inferences that can be supported by the representation may be matched appropriately with the tasks they support.

In order to design appropriate representations, we therefore need a calculus for the representing world, which defines the perceptual inferences which it supports. We will develop such a calculus in Chapter 4, when we consider two basic systems (topology and orientation), defined over binary relations between graphical objects, and derive rules for their combination.

Zadrozny considers a number of graph-based formalisms as representational systems capable of supporting reasoning:

> In a graph-based logic, syntax, proof theory and semantics (or model theory) would refer to graph-theoretic properties of terms that are manipulated ... to our best knowledge, there is yet no such graph-based logic. But we believe that a graph-based logic of concepts can be formulated.
>
> [163]

Graph-based formalisms are just one kind of visual representation. Zadrozny considers several formalisms with respect to the following aspects.

- **Syntactic:** the means by which the well-formedness of expressions may be determined.
- **Semantic:** the modelled world, its entities and relationships.
- **Proof-theoretic:** the inferences which are supported by the formalism.
- **Decidability and complexity:** how tractable are the operations to manipulate the objects ?

Harel [64] discusses one particular visual formalism, the higraph. This consists of *blobs* and *edges*. Edges connect blobs and blobs may contain sub-blobs. The formalism is equivalent to datalog, i.e. Prolog without negation, functions, recursion or variable sharing in rule bodies, which is known to be decidable. It may be given a semantic grounding in set-theory, with containment corresponding to set-inclusion. However, it cannot model certain forms of disjunction, in particular it cannot express the idea that some set is wholly included in one or other of two sets, without explicitly saying which set contains it: any element has to be depicted in a definite location. This system corresponds to the MARS of Stenning and Oberlander.

Visual representations frequently use absence of information to encode existential negation. Thus, a feature that is not explicit in the representation is assumed to be absent in the represented world. For example, in typical visual representations of graphs (network diagrams) it is generally assumed that where an arc does not exist in the diagram between two nodes, then those nodes are not connected in the domain. This form of representation by absence of information corresponds to the closed world condition of Levesque's criteria for a system to be vivid. This contrasts with standard logic-based representations of information in which nothing can be concluded from absence of information.

On the other hand, there are visual representations which represent only some part of the domain: nonetheless, within that subdomain the representation is complete and the absence of information is interpreted as negation. For example, in a tree-like diagram of a file system, some nodes (directories) may be shown 'expanded' and others 'closed'. The diagram is to be interpreted as meaning that, for the expanded nodes, all subdirectories and files are shown, and the absence of substructure in the diagram implies its absence in the corresponding domain object; on the other hand, nothing can be inferred about the absence of substructure in a closed node. In these cases, some form of ellipsis is generally used to indicate where information is lacking. For example, in certain versions of the Macintosh file manager (finder) open nodes are indicated by downward pointing arrowheads; in MSWindows file manager, closed nodes may be

indicated by a '+' sign on the relevant directory icon.

1.1.3 Visual representations as language

We now make a brief diversion to compare visual representations and text-based languages. Our characterization of visual representations as mappings between represented and representing worlds already bears many similarities to standard approaches to defining the semantics of languages.

In textual representations, it is usual for identity to be associated with terms, rather than with specific occurrences of those terms. Thus, multiple occurrences of a given term are taken to refer to the same object. By contrast, in many visual representations, visual objects with identical properties are not usually assumed to refer to the same domain object. In fact, it is often the case that there will be exactly one occurrence of the representing object for every represented object. In linguistic representations, each fact about an object is expressed by separate sentences, with separate occurrences of the object name. In visual representations it is possible for the same graphic object to participate in multiple relations with other objects and this is the usual way to express multiple facts about an object.

Visual representations allow the encoding of type and value information within the representation of the object. This has analogies in text-based languages: for example, the representation of a numeral encodes its value; or, as another example, some programming languages use naming conventions to convey information concerning the type of variables. In natural language, the use of modifiers gives information about number, role, tense and so on.

In visual representations referential identity depends as much on continuity of representation as on particular features of the representation: the representation of an object can change as its properties change, without implying any change in identity of the object. For this reason, in visual representations there is more scope to use the properties of the identifier to convey additional information about the object it represents. Shneiderman cites continuous representation as one of the characteristic features of direct manipulation [131]. In textual representations, the ability to convey information about the referenced object by modifications to the identifier is restricted by the requirement that the modified form be recognizable as a derivation from the same root (irregular verbs and plurals notwithstanding). Examples from natural language are the use of modifiers to indicate number (plural/singular) or tense; in written language, there are orthographic conventions which are used in an *ad hoc* way to give added meaning, such as

underlining for emphasis, or italicization of foreign or novel terminology; in programming languages orthographic conventions are often used to indicate object type, such as the use of an initial capital letter in Smalltalk to indicate a global variable, or the use of names beginning with letters from 'i' to 'n' in FORTRAN to indicate integer variables.

Chomsky has proposed a multi-layer model in which the elements of each layer are composed by an operation of concatenation to form the elements of the next [20]. The uppermost layer is the layer at which arbitrary statements of the language may be made. With written language, it is fairly easy to identify the different layers, with individual letters (the alphabet) forming the lowest layer.

Unfortunately, the number of 'alphabets' for visual representations is huge and many employ *ad hoc* conventions. International standards for symbology do exist, particularly where they relate to public information (road signs, health and safety markings and so on), although these are often adapted by local custom. The attempt to define an ISO standard for icons in computer systems was abandoned, although a *de facto* standard is emerging through usage and custom. Thus, we certainly cannot hope to define a universal alphabet for visual representations. The attempt to define the alphabet in terms of some set of geometric primitives is no more appropriate for visual representations than for textual ones: a *font* may be defined as a set of strokes and filled areas, but an individual character may appear in numerous fonts so that characterization of an alphabet in purely geometric terms is doomed to fail through excessive detail.

The problem, then, is to find ways to classify the range of possible graphical symbols, so as to make their large number manageable. Bertin has made a study of the semiotics of graphics, with particular reference to cartography [14]. The primitive objects and properties which he considers are

- **Marks:** points, lines & curves and regions.
- **Position:** in any of three dimensions.
- **Dynamic:** animated display of information.
- **Retinal:** size, shape, orientation, colour, pattern and saturation.

Bertin proposes a basic set of primitive languages based on these objects and properties and a composition algebra which is used to define new languages in terms of the basic set.

One problem is to distinguish features which count as symbols. This is related

to the use of the representation, as well as its formal semantics. Pratt discusses this issue in the context of defining a formal semantics for cartography. He uses the example of symbols representing land and water on maps, making the point that for some purposes it will be more convenient to treat land as symbol and water as background and vice-versa for other purposes [118]. In either case, it is possible to define a formal semantics for the map, and indeed it is also possible to define a semantics which accounts for both cases at once, but the 'natural' choice of a formal semantics is heavily influenced by the intended use of the map, since least redundant approaches which make reference only to the objects of interest in the domain are to be preferred. Of course, maps are extremely rich visual representations, capable of being adapted to a variety of purposes and they contain a number of different systems for encoding information. Indeed, it was this observation that motivated Bertin's approach of defining elementary visual languages, together with rules for their composition.

A simple view of the syntax of visual representations is that the basic unit of information is a pair of symbol and location [130]. The syntax defines rules for creating legal sentences of the language. These rules govern properties of the symbols and relationships between their locations.

Mackinlay describes a number of primitive visual language types based on this idea [97]. Some examples of these language types are

- **Single position:** location of marks along a horizontal or vertical axis.
- **Apposed position:** the location of marks along both horizontal and vertical axes.
- **Retinal-list:** which use Bertin's retinal properties to encode information in marks.
- **Connection:** in which marks are related by connecting them with lines.

The language types may be characterized in terms of three sets: the marks and their properties; the horizontal encoding; and the vertical encoding. Three composition operators are available for composing instances of the language types into higher-order representations.

- **Single axis composition:** two single-position languages are composed by merging their axes and presenting the marks from each language in parallel, disjoint alignment.
- **Double axis composition:** two apposed position languages are combined by overlaying their axes and marks.
- **Mark composition:** this allows the composition of different types of language. Marks are composed by merging the mark sets of the composites, wherever they encode the same information. This is achieved by combining the retinal properties of the marks.

In this thesis, we will take a different approach: rather than characterize visual representations in terms of the location of marks, we characterize the spatial relations between them. We do not consider any visual properties other than spatial relations, although, clearly, other properties such as shape, texture and colour are all important means of conveying information in visual representations.

In the cases which Mackinlay considers, the shape of marks is used to distinguish different categories of information. The position and size of marks are used to carry the information itself. Variations in shape within a given mark type (other than those which result from scaling) are not permitted. Our aim is to be able to describe visual representations in terms of a calculus of spatial relations. The lowest level entities in such a description are the basic symbols of the representation (Mackinlay's marks) and the spatial relations between them. When we come to choose the basic set of spatial relations (which we will do in Chapter 4), we will need to be careful that we make as few assumptions as possible about the topology and geometry of the basic symbols, because we cannot say in advance what those symbols might be. In so doing, we will ensure that the relations which we develop can be applied to any set of visual objects.

Our starting point is a given set of uninterpreted basic regions to which the relations developed in Chapter 4 may be applied. Our aim is not to provide a general method for description of arbitrary regions, rather to show how, given a set of basic symbols and spatial relations, we may describe a wide range of visual representations. Therefore we do not need to characterize the regions in terms of more primitive geometric entities. This contrasts with the approach generally adopted in constraint-based descriptions of the user interface, where the relations between regions are defined in terms of quantitative constraints (i.e. constraints based on arithmetic equations and inequations) between a set of geometric primitives (line, polygon, rectangle and so on).

Of course, in order to be able to render the representation on screen it will be necessary to provide a description of the geometry somewhere within the user interface and its associated toolkits: however, for the purposes of describing many of the properties of the representation it is possible to abstract away from this level of detail.

1.1.4 The user interface and conventional diagrams

A comparison between conventional diagrams and the user interface is useful, because it illuminates some of the difficulties associated with defining the correspondence between interface state and the represented state.

There are obvious similarities. Both diagrams and the user interface have an underlying 'alphabet' of basic symbols. They are structured into complex units by means of spatial relations between the entities (the syntax of the diagram). Modifiers may be used to add nuances of meaning to the symbols, such as when subgroups of otherwise identical symbols are distinguished by means of colour.

We can define an abstract syntax for diagrams in much the same way as for text-based languages. The interior nodes of the tree represent spatial relations and the leaves are individual symbols, together with their attributes (colour, size and so on).

However, it is problematic to apply this kind of analysis directly to the user interface. One difficulty is the dynamic nature of the user interface. The structure of a diagram is static and because of this relations between objects in the diagram are fully determined and immutable. A naive approach would be to consider the user interface as a succession of diagrams, each instantaneous state of the interface being treated as a separate diagram. However, the problem with this is that, even in very simple interfaces, the instantaneous state of the interface is ambiguous. Of course, such ambiguity may also be present in a diagram, but the art of presenting information diagrammatically is to design the diagram so as to remove ambiguities. The designer of the user interface has to allow for the fact that the user has a certain freedom to alter the layout of the diagram, which may inadvertently introduce ambiguity. The following example should clarify this point.

Suppose we wish to describe an editor for directed graphs in which nodes, represented by circles, are connected by arcs, represented by arrows. We will suppose that the user is free to alter the layout of the diagram by moving the nodes and that the interface automatically maintains the connectivity of arcs and

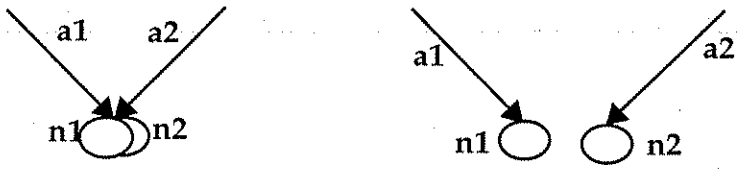
nodes. The domain relation between an arc and a node, that the arc is *incoming* to the node is represented (say) by the spatial relation of overlapping between the circle and the tip of the arrow. We will represent this relation as {overlap}, using the braces to remind ourselves that the allowed relation may be a set of primitive spatial relations. (In Chapter 5 we will develop a full syntax for the representation of spatial relations.) That is to say, the allowed spatial relations between arrow and node consist of the single relation{overlap} and all other relations are forbidden. The relation {overlap, disjoint} between arrow and circle has a different interpretation (that the arc and node are not connected). To the user, this distinction may be made apparent when the circles and arrows representing nodes and arcs are moved: in the former case, the arrow remains attached to the node when it moves, in the latter the node may become disjoint from the arrow when it moves.

Consider the situation depicted in Figure 1.1. It shows a 'before' and 'after' state of the direct manipulation graph editor. In part (a) of the figure, the relation {overlap} holds between a1 and n1, a2 and n2, a1 and n2, and a2 and n1. However, further interaction reveals that the relations shown in (b) are also allowed. These differ in the relation between a2 and n1, and that between a1 and n2 is {disjoint}. Thus the full set of allowed relations is

{overlaps} (a1,n1)
{overlaps} (a2,n2)
{overlaps,disjoint} (a2,n1)
{overlaps,disjoint} (a1,n2)

That is to say, the relation between a2 and n1 differs from that between a2 and n2 in that it is less constrained. At any moment, only one of the allowed relations will hold between two given objects. The full set of allowed relations between objects becomes apparent to the user through interaction with the interface. So the state of the representation is modelled by describing the *permissible* relations between interface objects, as well as the *actual* relations which hold at any instant.

One might argue that all interfaces should be designed so as to exclude such ambiguities, so that in the above example it would not be permitted to create the ambiguous state shown in part (a) of the figure. This may or may not be a good design principle: the usability of such a restricted system might be improved by the reduction in ambiguity, or it might be worsened by the resulting restriction on the user's ability to control layout. One could only decide this question through



(a) Before interaction **(b) After interaction**

Figure 1.1 A potentially ambiguous representation (a) which may be disambiguated through user interaction (b).

experimental evaluation. However, since our project is to be able to describe existing interfaces and, for good or bad, these *do* contain such ambiguities, we will need to select a form of representation which allows us to define permissible as well as actual states of the interface. We will show in later chapters that a representation based on qualitative constraints satisfies this requirement.

Another way in which the user interface differs from ordinary diagrams is in the fact that it is interactive. Thus it serves not only to inform the user of the state of the underlying system, but also to allow the user to change that underlying state. Consequently, some of the structures that appear in the representation relate to the process of interaction, rather than to the underlying domain model. A close parallel to such features in a diagrammatic context is the use of construction lines in technical drawing. The user interface is perhaps better compared with an *incomplete* diagram which the user is editing. Any instantaneous state of the display may show the traces of incomplete actions and provisional information concerning the domain model. Thus, whatever system of representation we adopt, it will be necessary to represent incomplete information arising from partially completed actions. We will see in Chapter 2 that this requirement has been highly influential in the development of theoretical architectures for user interface systems, where it has been referred to as the problem of providing *fine-grained semantic feedback*. (see, e.g. section 2.2.1). The architectures that have been proposed to satisfy this requirement add channels of communication, or purpose-specific layers, solely to handle feedback relating to partially completed actions. In the framework which is proposed in this thesis, the idea of working with partial information is fundamental to the notion of a constraint, so that we can handle incomplete information without recourse to special-purpose components in the architecture.

In order to describe properties of visual representations, we need more than a calculus of spatial relations: it is also necessary to model the structure of composite objects. Composite objects may be structures, sets or aggregates. A structured object is one which has named parts that have some functional relationship with the whole. Sets of objects are composites whose subcomponents are not distinguished functionally. Aggregates are objects whose identity and properties are derived from a set of components.

Structural relations describe how composite objects are composed from subcomponents. That is to say, they describe part–whole relations. The study of such relations is complicated by issues of object identity. In particular, composite objects may be identified purely in terms of their parts, so that two composites with identical parts are treated as identical, or they may be ascribed an independent identity such that two compounds with identical parts can be non-identical.

Another issue is that of the independent existence of the parts: in some cases the parts may be inseparable from the whole, incapable of independent existence; in others it is possible for the part to exist on its own.

Formal models of part–whole relations originate with Husserl and have been debated widely since then (a good review of such theories is presented by Simons [132]). This thesis does not aim to give a formal ontology for user interfaces. rather our aim is to describe an approach which has practical application in a variety of contexts. However, structural relations between graphical objects are fundamental to the description of the user interface.

Possible structural relations are

- **Named parts:** These are subcomponents of an object which have an independent existence as objects in their own right. They can be identified with respect to the composite by means of a labelling function. The composite itself has an identity that is independent of the identity of its subcomponents.
- **Set member:** The subcomponents are members of a set. They can be identified either severally or individually be universal and existential quantification with respect to the set and also retain their identity when removed from the set.
- **Aggregate parts:** In contrast with proper parts, aggregate parts are not distinguished subcomponents. Thus a region has as its aggregate parts all those regions which it contains; a set has its subsets as aggregate parts; a text those words and characters of which it is composed. Aggregate parts do not have an independent existence.
- **Properties:** Properties have it in common with named parts that they can be identified by means of a labelling function, but unlike named parts, they do not retain their identity independently of the object of which they are part. In this sense, they resemble aggregate parts.

The interpretation of pictures depends in part on the use of a set of pictorial conventions, specific to some domain of representation and in part on the ability to distinguish a set of primitive relations and properties employed by the conventions. Pictorial objects are structured sets of such primitives and their conventional meaning is derived from relations existing between them.

One of the problems of defining a semantics for pictorial languages is that of identifying the syntactic elements in any given representation. In a conventional language, the syntactic structure of expressions is tree-like: every element is a direct component of exactly one structure. Semantically, this generally is not the case: the same semantic object may be represented by multiple occurrences of syntactic objects. In pictorial languages, there tends to be a one-to-one relationship between syntactic and semantic objects, with the same syntactic object (icon, symbol or whatever it may be) participating in multiple structures.

1.1.5 Cognitive aspects of visual representations

Precise description of visual representations needs a syntactic and semantic basis. The syntactic basis identifies an alphabet (or alphabets) of primitive elements and rules for their composition. The semantic basis states how the syntactic structures may be related to properties and relations in the represented world. The discussion above of homomorphic systems, MARS and the like suggests that there are advantages to having a simple mapping from syntax to

semantics in which there are one-to-one correspondences between syntactic structures and the objects they represent. However, acceptance of this supposition still leaves a number of hard questions to be answered.

Some of these questions are psychological in nature, e.g.

- Is there a cognitive basis for choosing some particular alphabet of visual features and composition relations ?
- Or is the 'best' set determined *solely* by usage and culture ?
- Can there be a universal set of elements, suitable for defining any visual representation ?
- How does the choice of syntactic elements influence the effectiveness (ease of use) of the resulting representation ?

In order to address these questions it is necessary to consider the nature of the internal representations used by humans for reasoning about and understanding the world: this is a perennial source of controversy among philosophers and psychologists. Berkeley [86], Kant[86], Galton[55], Anderson [4] & Jackendoff [80] are but a very few of those who have made notable contributions. We will not enter into this thorny debate and make no attempt to resolve any of these issues in this thesis.

Nonetheless, if any kind of formal account of visual representations is to be given, it is necessary to have a foundation of some kind. Our starting point is to define a set of spatial relations that is computationally tractable and then show by examples that the resulting system may be used for description of typical user interfaces. It would of course be satisfying if we could also show a psychological rationale for our choice of basic relations, but this will have to be the subject of future research.

1.2 The thesis

We now give a summary of the position which this thesis puts forward. The general area of interest is, of course, user interface construction and the specific area of interest is the use of qualitative spatial constraints in the specification and construction of user interfaces. The thesis makes the claim that certain qualitative techniques of constraint solution have a useful role to play in user interface construction. These techniques have not hitherto been applied to this specific problem, although they have been discussed in other contexts. The detailed exposition of the thesis follows several related strands of argument.

We will consider the limitations of those techniques of constraint solution which have been applied to the problems of user interface construction.

The qualitative techniques which are proposed need some adaptation to be applied in the context of user interface construction. This is because they originate from a problem-solving context in which it is desired to find a 'best' solution to a set of initial conditions. We will use them to describe continuous dynamic behaviour of the user interface, rather than to achieve a single best solution.

In order to show how the techniques may be applied to practical cases, we develop a notation for the description of user interfaces. We are conscious of the fact that notations have different roles and that there is often a compromise between the flexibility of expression which is required of a notation intended to support informal communication between designers and users and the semantic well-foundedness which is required of a notation which is to be used as the basis for formal reasoning. Much of the thesis is concerned with giving the notation a formal semantics. To achieve this formalization, we employ a calculus of spatial relations and a model of computation with constraints.

The thesis embodies a model of the user interface. It is not a claim of the thesis that this model is the best or only one possible. However, we mention it now in the belief that it will help the reader to understand the relationship between the various strands of the thesis. In this view, the user interface is a representation of the state of some system modelled in the computer. We will be concerned with user interfaces in which the representation is predominantly visual, using spatial relationships between objects to represent aspects of the underlying state.

As such, there are properties of the interface and user interaction which may only be defined, described and reasoned about in terms of spatial relations between objects in the interface. However, the view of the user interface as a visual representation needs some qualification because of its dynamic character. Approaches to defining a semantics for visual representations which are based on a linguistic model often seek to define a meaning function which maps from static visual structures to structures in the represented world. Such an approach, applied to the user interface, would attempt to define a mapping between the instantaneous state of the interface and the state of the underlying model. However, this tends to produce highly moded mappings. Moreover, there is generally a high degree of redundancy in the representation. We will argue that the 'meaning' of the user interface may be derived not from its instantaneous state but from the set of states which are permitted at any point. Of course, the user at any moment can only see directly some part of one of the many possible states:

much user interaction is concerned with exploration of the allowed states of the interface, rather than with performance of specific application tasks.

We shall formalize a computational system in which the atomic values are regions in a 2.5D space and over which we can apply binary constraints representing spatial relations between objects. We will develop an agent-based notation which allows us to represent the set of possible display states using constraints and describe a computational model which allows the display state to be shared with external concurrently executing processes.

This view contrasts with other agent-based approaches in which the focus is on interpreting low-level input actions performed by the user and relating them to events and actions in the functional core [1,34,40,66]. In such approaches there is no explicit formalization of how spatial relations between objects in the interface are to be interpreted in terms of the state and behaviour of the functional core. In these approaches, the most that can be formalized of the spatial properties of the interface is the *context* of any input action (e.g. the location of a pointing device).

We derive a calculus of spatial relations in 2.5D and show how it can be embedded into a computational model based on agents. The calculus and computational model, taken together, give a formal semantics to a notation which we present for the description of the user interface as a visual representation.

1.3 Structure of the thesis

The previous sections have described the background to the thesis. We now give a brief summary of the remainder of the thesis, chapter by chapter.

1.3.1 Chapter 2

In Chapter 2, we look at the general requirements for systems which support the design and implementation of highly graphical user interfaces to computer systems. The term 'highly graphical' refers to the use of visual representations, such as described above, as a primary means of manipulating and conveying information.

The task of developing graphical interfaces has proved to be difficult and costly, so that much research has been devoted to finding ways to support the production of such systems, both at the level of specification and of implementation. This research has been directed mainly towards discovering techniques for constructing user interfaces out of modular components and solving the architectural problems which arise in this pursuit. Very little research

has been addressed to the specific issue of providing abstractions for the visual representations to be found in the user interface. There is a tendency to approach this issue from the viewpoint of system architecture, concentrating on issues of control and communication, rather than on visual properties of the interface. Without doubt, control and communication are important aspects of the problem and several models of the user interface have been developed to describe these processes. Indeed, we will refer frequently to one of them (the 'arch' model) as a means to structure the discussion. However, we will argue that the failure to consider the nature of visual representations has led to systems which lack the expressiveness required for the demands of developing highly graphical user interfaces.

1.3.2 Chapter 3

Chapter 3 discusses the properties of constraint programming which make it a suitable candidate for implementing highly graphical user interfaces. We review some of the algorithmic techniques for solving systems of constraints and consider the issues involved in incorporating constraint-solving techniques into programming languages. Constraint-solving techniques have been applied to the construction of user interfaces but these have employed quantitative methods directed towards finding numerical solutions of constraints on point locations. We discuss how this approach limits the expressiveness and effectiveness of the resulting languages and tools. We then consider the argument for the application of qualitative formulations of constraints to user-interface construction.

We consider the solubility of systems of qualitative constraints as a preface to the exposition in Chapter 4 of a system of spatial relations.

1.3.3 Chapter 4

In Chapter 4 we review various approaches to the definition of calculi of spatial relations. Some of these are axiomatized by geometric theorems, others (the *mereological* approaches) proceed from a given, uninterpreted relation between regions. We put the case that this latter kind of approach suits our purposes better, since there are geometric anomalies in the 'two-and-a-half dimensional' virtual space of overlapping windows and flat regions in parallel planes which characterizes many graphical user interfaces. We show how a mereological approach may be used to define systems of spatial relations and then we develop calculi for reasoning about topology and orientation. Finally we use the results of Chapter 3 to show that both systems have polynomial time

decision algorithms.

We additionally show that the mereological approach can, in principle be extended for reasoning about edge contact, shape and size, although we do not develop these systems any further in the thesis.

1.3.4 Chapter 5

In chapter 5 we show how the user interface may be modelled as a set of concurrently executing agents with communication mediated by a constraint store. This model of computation has been termed *concurrent constraint programming*. The model is often referred to as the cc() framework, since a whole family of concurrent constraint programming languages may be derived from it, with each member of the family characterized by the set of basic operations which it supports. In chapter 5 we present a notation for describing spatial relations in user interfaces and relate it to the cc() framework.

The framework has two basic operations, ask and tell, which respectively check whether a constraint is entailed by those in the store and augment the store with a given constraint.

We give formal descriptions of the constraint store, its operations and agents and then present a constraint programming language based on them.

1.3.5 Chapter 6

This chapter contains examples which illustrate the use of the language that is developed in Chapter 5. The aim of this chapter is to show that a number of frequently occurring idioms in direct manipulation interfaces may be expressed simply within the notation.

We additionally extend the notation by providing shorthand for useful idioms. These extensions do not extend the semantic interpretation of the language and may all be expressed as constructs within the basic language.

1.3.6 Chapter 7

In this chapter we put the research findings of the thesis into context by comparing them with related work and by considering possible extensions of the research carried out for the thesis.

We compare the interface model presented in this thesis with other systems based on constraints.

1.3.7 Chapter 8

Finally we give a summary of the main research findings of the thesis.

Chapter 2 The user interface

This chapter reviews work in user interface management systems (UIMS), describes some important basic concepts and defines terminology that will be used in later chapters. The introduction in section 2.1 defines a basic vocabulary that will be used to discuss issues in UIMS. In section 2.2, we look at some models that have been proposed for describing UIMS and reasoning about their properties. Some actual systems are analysed using these models in section 2.3. In section 2.4 we consider the view that there is a need for abstractions of presentation aspects of the interface, and that, whereas the problem of providing abstractions of user input has been widely researched, there is very little work aimed at providing abstractions of visual structure. We illustrate the point by considering the limitations of the work to date in this direction. Finally, we summarize the argument with a check-list of requirements for user-interface architectures.

2.1 Introduction

The user interface supports users in performing tasks. It provides access to functionality in the underlying system and presents information needed by users for planning and performing tasks. It is a large and complex area for research: on the one hand, its concerns are related to system architectures, inter-process communication and software reuse and reliability; on the other they are cognitive, ergonomic and socio-cultural.

An important aspect of the user interface is the manner in which information is presented. The level of detail is crucial to the usability of the interface: too little and the users will be unable to perform their tasks, too much and they will be overwhelmed. The design of the notations used to present the information influences the way in which that information is understood and thereby determines how the system is used [61,93,119]. We reviewed some theoretical approaches to this problem in the previous chapter. In recent years there has been a tendency to use presentations of a pictorial nature, employing conventions from graphic design and diagrammatic techniques. The motivation for this trend is to improve the ease of use of the system.

Just as the user interface is complex, so is the task of specifying it. The role of the interface designer is to balance the needs of users against system capabilities; to reason about the users' tasks at multiple levels, cognitive, ergonomic and organizational; and to reason about the system at levels of specification, implementation and architecture in order to produce designs that yield reliable, usable systems.

This complexity can be reduced by distinguishing separate issues and dealing with them as far as possible in an independent way. Areas of concern that have attracted research attention include programming languages, architectural models, formal specification, representational systems, requirements capture, task analysis and evaluation techniques. The methodology of design is itself an area of research whose objects of study are systems for classifying and handling the various issues. Each of these areas of concern has its own complexities and problems, but the whole is further complicated by the need to integrate their disparate techniques and philosophies.

Design methodologies that call for iterated cycles of analysis, specification, implementation and evaluation are frequently employed as a means to control the complexity of the design problem. Such methodologies demand techniques of interface construction that allow easy modification and reuse of existing components. The search for these techniques has motivated much of the latest research into UIMS.

Green argues for the view that HCI is poor in vocabulary and that consequently it is hard for researchers to present their results concisely and unambiguously [62]. There is certainly a need for a general framework in which to discuss the user interface and in this chapter we review the research in this direction. In so doing we will introduce and define some vocabulary. This is not because of any shortage of terminology in the literature, rather that there is no consensus on vocabulary and different authors use the same terms in different ways. The process of defining terminology also serves to clarify the underlying concepts.

The starting point for our definition of terminology is that proposed by the UIMS Tool Developers Workshop [9] which has also been adopted by the IFIP Working Group 2.7. A brief summary of the most used terms follows.

- **Application:** the term is used in two distinct senses in the literature. Sometimes it is taken as referring to the whole system developed for end-users, at other times it refers to just that part of the system which implements non-interface code specific to the area of application of the system. The term is used here in the former sense. An interactive application is one which has a user interface.
- **Domain:** the domain, or application domain, is the area of activity for which the application is developed. Typical domains are bookkeeping, document preparation and network management. The domain is not system specific, unlike the functional core (see below).
- **Domain model:** the domain model is a system-specific model of those aspects of the domain which the application is designed to support. many authors use the term 'domain' in this sense.
- **Functional core:** the functional core is the implementation of routines and objects that are specific to the application and which perform or support some of the tasks in the domain. As noted above, the term 'application' is used by some authors to refer to the functional core.
- **Interface toolkit:** a collection of object classes that implement user input functions and presentation of information. Examples are OSF Motif ™, Apple Macintosh™ toolbox.
- **UIDE:** User Interface Development Environment is the total ensemble of tools and software for specifying, building and evaluating the user interface.
- **UIDT:** User Interface Development Tools are specialized tools for handling particular aspects of interface design.
- **UIRS:** User Interface Runtime System is the runtime environment that supports the user interface.
- **UIMS:** User Interface Management System is the assemblage of UIDE, UIDT, UIRS, although some authors use it in the very restricted sense of UIRS only.

Several roles can be distinguished for those taking part in constructing and

using interactive applications.

- **UIDT developer:** builds the tools that are used in the UIDE.
- **Interface developer:** builds the interface using UIDTs. The skills required of the interface developer will depend on the range of tools available. Often, interface developers need to have expert programming skills because of poor tool support.
- **Interface designer:** specifies the behaviour and appearance of the interface and its links to the functional core. This role is often combined with that of interface developer.
- **User:** this is the end-user of the interactive system.

2.2 Architectures for UIMS: a brief survey

Architectures describe the interrelation, communication and roles of the subcomponents of a system: *concrete architectures* describe particular implementations; *abstract architectures* are generic templates which may be instantiated by concrete architectures. A *reference model* is an abstract architecture which serves to identify essential functionality in the architecture, but which is not instantiated directly. Particular concrete architectures may be related to the reference architecture, however there will not necessarily be a one-to-one correspondence between the components of a reference architecture and those of the concrete architecture.

A reference model provides a means to categorize aspects of the design and implementation of the interactive system. Specific implementation architectures and design activities can be discussed with reference to the model, but the model does not prescribe any particular approach to these architectures and activities. Reference models have been widely accepted as a means of standardizing systems: for example, the seven-layer model of communication forms the basis of the OSI standard for communication [99]. There is as yet no standard reference model for UIMS, though several have been proposed and this is currently a research area [10].

Multi-agent architectures are architectures in which interaction is modelled by agents communicating through shared channels. These have become important recently because they support concurrency and distribution and have found applications in computer-supported cooperative working and distributed object-oriented systems. They also provide a natural means to structure multithreaded dialogues. When used as a tool for specification, multi-agent architectures are generally based on synchronous communication, in the style of Milner's CCS

[103]; when used for concrete implementations, the model of communication is generally asynchronous, in the style of Agha & Hewitt's actors [2].

A recurring theme in the literature on UIMS is separation of interface and functional core. Wood and Gray distinguish three possible forms of separation [160].

- **Conceptual separation**. The interface and the functional core are specified separately. The interface is described in terms of interaction objects and techniques, the functional core in terms of domain-specific operations and objects.
- **Methodological separation.** The development of interface and functional core proceed separately, as far as is possible.
- **Architectural separation.** The interface and functional core comprise separate modules.

These represent increasingly strong forms of separation: methodological separation implies conceptual separation; and architectural separation implies methodological separation.

One benefit of architectural separation is the possibility of modifying or exchanging the interface with little or no change to the functional core itself. Techniques for specifying graphical interfaces generally lead to very redundant descriptions, because much of the state of the interface has no semantic significance. For the same reason, the state of the functional core underdetermines the state of the interface. Formal approaches have been used to abstract away from this redundancy [66]. However, as Took [148,149] notes, many changes to the state of the interface have relevance to the user, but not to the underlying functional core. One example he gives is that of moving a dialogue box so as to uncover information hidden beneath it. The position of the dialogue box has no significance to the functional core, but is of relevance to the user. Another example is where the user rearranges the icons within a file-manager window. Provided the icons are not transferred to another window, there is no significance to the underlying file system, but the user may well find it convenient to group the icons in some particular way, such as to put the most frequently used ones at the top of the window. Took refers to such changes which relate to the presentation of the interface, but have no relevance to the underlying functional core as *surface interactions*. Deep interactions are those which are significant to the functional core. The ability to manage such surface behaviour is clearly important from the point of view of the usability of the system. If surface interactions are handled

from within the functional core, then this will add to the complexity of the functional core code. Moreover, most surface interactions are generic and to be found in a wide variety of interfaces, thus it makes sense to handle them separately within an architecture that allows for their reuse. In the context of the discussion of visual representations in Chapter 1, surface interactions are those which involve navigation through allowed instantiations of the interface. Green [62] proposes the use of structure maps, a variation of E-R diagrams, to make generic interactions explicit.

Given that some form of separation is possible and desirable, then the question naturally arises: What implications does this have for architectures for UIMS ?

2.2.1 Seeheim and separation

The Seeheim model is an early and influential attempt to define a separation between interface and functional core [60]. It is based on a linguistic model of interaction which defines lexical, syntactic and semantic components. In the Seeheim model, these are called the presentation, dialogue and application interface components, respectively. A fourth component allows direct communication from the application interface to the presentation component for the purpose of providing feedback information. This is under the control of the dialogue model, which can initiate feedback. These components together constitute the user interface and are linked to the functional core which consists of domain-specific routines and data. The original conception was an analogy with command languages in which tokenized streams of input (lexical component) are parsed (syntactic component) and executed as calls on routines in a functional core (semantic component). Output is passed back through the same pipeline. The results of the calls on the functional core (semantic) are formatted as textual representations (syntactic) and sent as character streams to the display (lexical) (see Figure 2.1).

The presentation component is concerned with the manipulation of a set of basic tokens, which provide control and communication with physical devices for input and output. The specification of the presentation component involves a definition of these tokens. The dialogue component handles the translation of tokens into appropriate calls on the application linkage. The interpretation of the presentation tokens is almost always context-dependent, so that the dialogue component needs to maintain information related to the current state of the

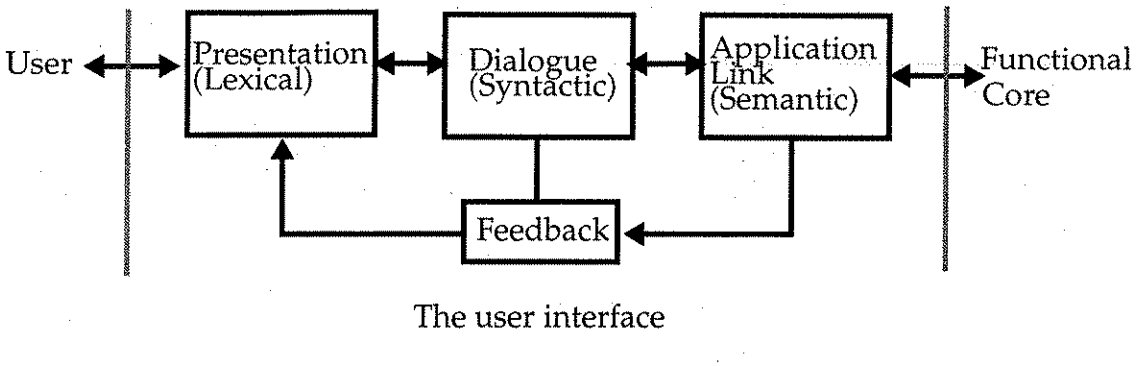


Figure 2.1 The Seeheim model of the user-interface, showing the correspondence with the three-layer linguistic model.

dialogue. The application link should not be confused with the functional core itself: the role of the application link is to convert data supplied by the dialogue component into a format suitable for passing to the functional core, providing default values as necessary and possibly calling several functional core routines to effect a single command. In short, the application link provides the user's view of the functional core.

The first attempts to define semantics for graphical user interfaces used a linguistic paradigm in the style of the Seeheim model. Such paradigms consider three layers: lexical, syntactical and semantic. Descriptions of systems which use this kind of modelling make a clear separation between specifications of input and output. Input is modelled as the interpretation of sequences of tokens from devices such as mouse and keyboard; output is modelled as operations on structured graphical objects. A typical description of an interactive user-interface might be structured as in Table 2.1 below.

	Input	**Output**
Lexical	mouse movements and actions	drawing primitives
Syntactic	gestures (drag, drop etc.) ; operations applied to graphical objects	structured graphical objects; widgets
Semantic	functional-core callbacks	high-level drawing routines

Table 2.1 The typical structure of 'linguistic' descriptions of the user interface.

The linguistic model was derived from methods used to specify command-line systems. In such systems there is generally a clear distinction between the

syntax of input and output operations. It is not normally the case that output from one operation can be interpreted as a command (one exception is the case of natural language interfaces). It is interesting that this distinction has carried over into descriptions of graphical user-interfaces, where it leads to a number of problems related to the difficulty of establishing the correct context in which to interpret the input actions.

Shneiderman has characterized direct manipulation as '... the continuous representation of objects of interest'[131]. The user has the impression of interacting directly with the display objects, of being able to pick them up, move them, stretch them and so on; some of these actions will have a significance to the functional core, others will result in changes to the display, but without any effect on the functional core. The output from the functional core also results in the same operations: objects are moved, resized and so on. The interface is a representation of objects in the functional core and operations on the interface objects correspond to operations in the functional core. Changes to the objects in the functional core are reflected by changes to the display objects; conversely, the same changes to the display may cause the changes in the functional core.

A simple example is a slider which controls a value: the position of the slider has a relationship with a value in the functional core that can serve either as output, when that value is changed from within the functional core, or as input, when the user sets the value by moving the slider knob. The semantics of the slider is thus very simple: its model is a range of values and operations on the slider denote particular values in the range.

Practical examples of sliders achieve the necessary abstraction in various ways. In one form of abstraction, the input operations are considered as acting upon a value, rather than on the 'thumb' of the slider. Changes to the underlying value are then reflected in the position of the thumb. In this formulation, input operations are abstracted as operations on the value, increment, decrement and so on. However, the semantics of these abstract inputs are heavily dependent on the graphical properties of the slider: the position of the mouse-pointer relative to the slider scale must be known in order to determine the change in the underlying value. Modification of the behaviour of the slider will be made more complicated by the close coupling between graphics, input operations and value.

A second way to describe the semantics of the slider is to relate the display to the value using a bidirectional constraint. In this case, generic definitions of input

operations such as 'drag' may be used without the need to define their relation to the semantics of the slider. The particular sequence of input operations that the user performs to move the thumb is irrelevant to the way in which it relates to the functional core: the user might use arrow keys or mouse movements, for example. However, it is also necessary to employ constraints that relate purely to the display aspects of the slider: e.g. the thumb is constrained to move in a vertical direction only.

The slider also gives an abstraction for functional core semantics: that is, it can be abstracted as the value which it represents. It is this form of abstraction that enables the slider to be interchangeable and pluggable as a widget. However, most widget sets provide a very poor set of semantic constructs, restricted to setting values and initiating call-back functions. Model-based systems such as UIDE [43] provide the means to represent, and reason about, a wider set of functional core semantics.

This simple three-layer linguistic model was not intended as a concrete architecture. It offers no support for structuring the three components and there is no support for concurrency, which has to be built in to the dialogue structure in *ad hoc* ways. The model provides no clear separation of the roles of the three components and the problem of providing 'fine-grained semantic feedback' has been cited as a case in which there is an overlap of presentation and application interface responsibilities. We referred to this point in section 1.1.4.

2.2.2 Homogeneous object spaces

The main contribution of the Seeheim model has been as a reference model which has stimulated research into particular notations and concrete architectures. Seeheim does not model the tool-rich environment of UIDEs, where, for example, the presentation component may be implemented partly using specific toolkits and partly using generic abstractions. Thus, it is often required to implement a system that can be used on different platforms, with different underlying interaction toolkits. It is desirable to be able to do this without redesigning the whole of the presentation layer, but Seeheim offers no way to manage the toolkit independently of the abstract description of the presentation. Related to this is the question of how to provide mechanisms for modularity and composition of interface objects.

An alternative approach to Seeheim is to model the interface as a set of objects all having the same basic structure. Dance *et al* refer to this as an *homogeneous*

object space [32]. Such approaches address directly the issue of composition of interface objects.

Two important models which fall into this category are MVC [90] and PAC [29]. These are illustrated in Figure 2.2.

MVC is the user-interface architecture of the Smalltalk-80 programming environment and as such is a concrete architecture. It distinguishes three classes of component: the Model, the View and the Controller. The model corresponds to the functional core of Seeheim. The controller is responsible for initiating actions in the model, dialogue control and handling user input and thus incorporates aspects of all three components of Seeheim. The view is responsible for output alone and is therefore a subset of the presentation component of Seeheim. The model component is shared by multiple views, organized hierarchically within windows, with each view having a single controller.

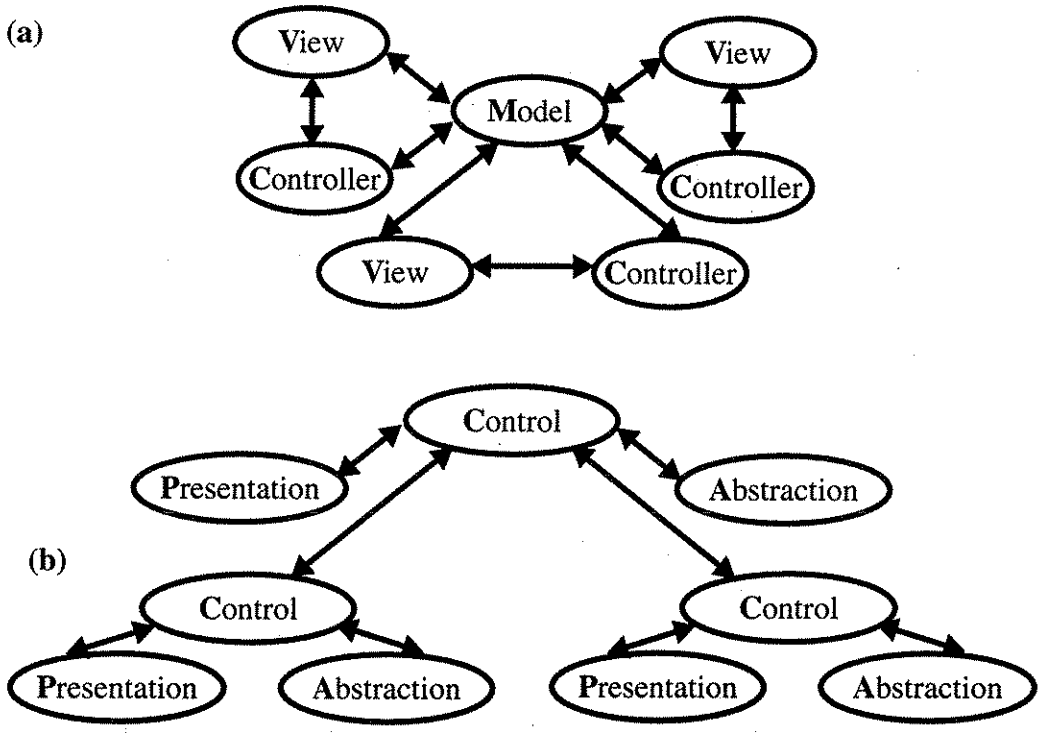


Figure 2.2 Control and communication (a) in MVC (after Krasner & Pope [90]), showing multiple view-controller pairs attached to a single model; and (b) in PAC (after Coutaz [29]), showing the possibility of hierarchical composition of objects through the control component

Modularity and composition are achieved by 'plugging in' different views to

the model. In order to build interfaces with MVC, the designer has to construct view-controller hierarchies, connected to particular aspects of the model.

In the original implementation of MVC, view-controller pairs were created by subclassing and the model retained no view- or controller-related information. It simply maintained a list of its dependents which it notified following any change. However, subsequent implementations used the notion of 'pluggable views', in which common behaviour was factored into generic view classes. A pluggable view is instantiated by parameterized method calls. For example, a number of views may involve the selection of an item from a list and differ only in respect of what is done with the selection. Rather than use subclassing to create new MVC objects, the generic list view can be parameterized with a method name (procedure) to be invoked in the model following selection of an item. This approach has two advantages: it allows dynamic configuration of the interface; and it reduces the amount of coding. However, it also increases the coupling between the components, by requiring the model to retain information that would otherwise reside in the controller. In the example of the generic list view, the model needs to keep track of the current selection, so that it can respond appropriately to the method call.

A second example of the way in which the distinction between view and controller becomes blurred in MVC is the case of pop-up menus. In some implementations of MVC, display characteristics of pop-up menus are handled by the controller; in others they are views in their own right. So, whereas MVC improves upon the naive linguistic model by offering mechanisms for modularity and composition of components, it does not support the designer in deciding how to allocate functionality between the three layers.

View–Controller pairs in MVC may be structured into a view hierarchy, with control being passed down the hierarchy to subcomponents of the view, but in the original version of MVC, no support was available for a similar structuring of the model. Later versions of Smalltalk remedy this by introducing an additional component, the **Application Model**, which allows for the coordination of multiple models in the same interface.

PAC (**P**resentation, **A**bstraction, **C**ontrol) is similar to MVC, but organizes the control and communication differently. In PAC, the equivalent of the model of MVC is the abstraction component; however, the presentation component of PAC handles both input and output. In PAC the control component does not handle

user input. Whereas in MVC the unit for hierarchical composition is view-controller pairs communicating via a shared model, in the PAC architecture, the unit of composition is PAC triads, whose control components are organized into a hierarchy. In MVC the views are able to communicate directly with the model; in PAC the presentation components can only communicate with the abstraction indirectly, via the control component. The rationale behind PAC is to enforce the decoupling of presentation and functional core information. Modularity in PAC is achieved by combining self-contained PAC objects, which contrasts with MVC in which modularity is at the level of view-controller pairs. The advantage of this approach is to hide low-level aspects of the functional core. Since PAC is not associated with a particular implementation, it is best considered as an abstract architecture.

2.2.3 The arch model

As a reference model, the Seeheim model has been influential in its emphasis on the separation of presentation, dialogue and functional core concerns. The issues of modularity and separation of roles have been addressed in the 'arch model', a modification of Seeheim which provides additional linkage components to act as adaptors for modules [8,10]. This results in a five-layer model which is intended exclusively as a reference architecture (see Figure 2.3). Particular concrete architectures will have differing boundaries between presentation and functional core. The terminology used varies somewhat between publications, although the basic concepts are unchanged. The description used here takes its terminology from [10].

The five components of the arch model are as follows:

- **The interaction toolkit (ITC)** provides a set of objects for handling input and output.
- **The presentation component (PC)** provides linkage between the dialogue component and the interaction component and contains abstractions whose role is to decouple those two components. Its purpose is to provide a set of presentation objects that communicate with the dialogue component and which are implemented concretely by the toolkit in the interaction component. Presentation objects are abstract interaction objects that are independent of the implementation of the interaction toolkit. In contrast, the interaction toolkit itself is platform- and implementation-dependent.
- **The dialogue component (DC)** sequences operations at the task level and is responsible for coordinating the interaction toolkit and functional core. The dialogue component ensures that interaction objects

are made available as appropriate to the task and organizes the users' inputs into formats suitable for passing to the functional core. It maps operations at the interface onto ones in the functional core. Additionally it is responsible for the coordination of multiple tasks running in parallel.

- **The functional core adaptor (FCA)** links the functional core to the dialogue component. Its purpose is to provide support for aspects of the dialogue that have no direct counterpart in the functional core. It may be used to provide semantic feedback by acting as a proxy for objects in the functional core. It is analogous to the presentation component in that it provides abstractions (conceptual objects) in order to decouple the dialogue component from the functional core.
- **The functional core component (FCC)** controls the application and provides access to base domain data and functionality.

The arch model expresses control and communication structures in a similar way to PAC. Modules may be combined by sharing of dialogue components. This allows for multiple interaction toolkits to be connected to multiple functional cores. The dialogue component handles all communication with other modules. This is illustrated in Figure 2.3.

The arch model will be used in this and subsequent chapters as a reference model for identifying concerns in the construction of user interfaces.

The arch model views interactive applications as being composed of a functional core and a user interface. Within the user interface there are four categories of object.

- **Domain objects** are domain-specific data structures and operations. They may not correspond one-to-one with data and operations in the functional core, but there is a well-defined mapping from domain objects onto functional core routines and data. The domain model is described in terms of domain objects.
- **Conceptual objects** are abstractions of the domain semantics which the user manipulates through interaction with the interface. Conceptual objects provide an abstraction of domain-specific data structures and operations. The handling of conceptual objects will be depend-

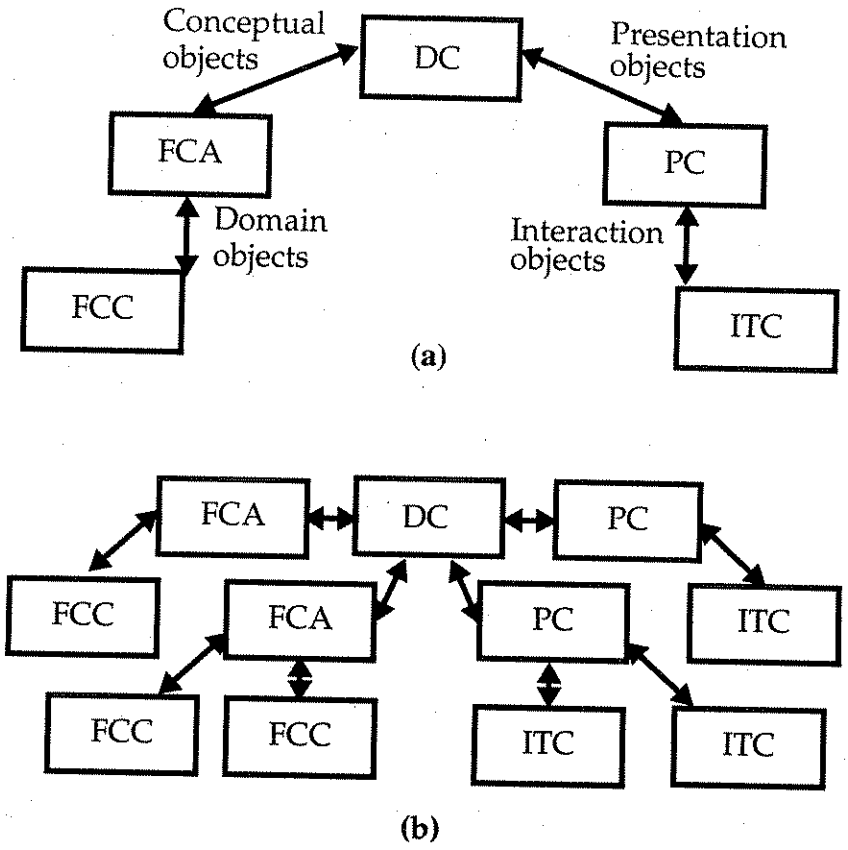


Figure 2.3 The arch model: (a) the components and their interfaces; (b) composition of components (after Bass *et al* [8]). DC = Dialogue component; FCA = Functional Core Adaptor; FCC = Functional Core Component; PC = Presentation Component; ITC = Interaction Toolkit component.

ent on the state of the system. Sometimes the functional core adaptor component may use them as proxies for the functional core (see above), at others it may convert them to sets of domain objects.

- **Presentation objects** abstract away from features of the presentation, in the same way that conceptual objects abstract away from the functional core.
- **Interaction objects** are implementation-dependent toolkit components, with which the user can interact directly. They are used for the implementation of presentation objects.

These categories of object each have responsibility for communication between two of the five categories of component in the arch model, as is illustrated in Figure 2.3 (a).

It is worth stressing again that the arch model is proposed as a reference architecture. This means that it can be used to describe a variety of concrete

architectures: the components of the model identify functionality in the concrete architecture under consideration, but do not necessarily correspond to distinct modules in the concrete architecture.

2.2.4 Agents

One approach to modelling the user interface is as a system of communicating interactive processes or agents. This is a natural way to specify multithreaded interactive systems where the behaviour of the system is conceived in terms of simultaneously executed threads of control. Agents can also serve as a basis for implementation of such systems, though in this case considerations of efficiency must be weighed. Modularity and composition are easily expressed in agent-based systems, as are certain dynamic properties of interaction, such as dialogue control.

It is important to note that most research into agents used as a tool for specification uses a synchronous model of communication, whereas research into agents used as a tool for implementation generally uses an asynchronous model. (In synchronous communication, sending and receiving of messages does not occur until both sender and receiver are ready. In asynchronous communication the receiver has no control over when messages are sent.) To some extent this difference in approach may be accounted for by the fact that in specification the aim is to model observable properties of systems. In these situations a synchronous model of communication has the advantage that it avoids problems associated with non-determinism in the order in which communications arrive. On the other hand, the implementation of asynchronous communication is much simpler and may be handled efficiently by such devices as message queues.

From the point of view of reasoning about formal properties of systems, the synchronous model has the advantage that there are well-understood calculi derived from it. This means that it is possible to derive properties of composite agents from the properties of their components. Unfortunately, it is much harder to define such calculi in the asynchronous model and although there have been several experimental implementations of agent-like UIMS using asynchronous communication, none of these has a formal semantics which supports reasoning about properties of agents. Those implementations of agents (e.g. Janus [83]) which do have an explicit formal semantics do not offer direct support for the construction of interfaces.

Agent-based architectures address different issues from those based directly

on Seeheim. The latter place emphasis on distinguishing separate concerns in the design, but have little to say about how they should be structured internally or how they should interact with each other. Agent-based architectures concentrate instead on communication and control, without enforcing any policy about separating out functional core from presentation and dialogue.

Agents have been widely used for formal specification of interaction and there is a large body of theoretical research on the formal properties of agents that can be applied to the analysis of interactive systems. Process algebras such as CCS [103] and CSP [76] lend themselves readily to specification of dialogues and allow the formal demonstration of properties of systems, independently of their construction.

This is not to say that these formal techniques preclude the need for experimental evaluations of interactive systems: there are many aspects of such systems that cannot adequately be captured in the formalism. For example, they do not make predictions about the usability of particular visual properties of displays. In terms of the arch model, they deal with the dialogue component. Formal description of a system is not enough in itself to ensure good design, although it may provide valuable corroboration and validation of the designer's intuitions, as well as indicating areas of possible difficulty for the user.

The appropriateness of the presentation aspects of the interface for a particular user depend on cognitive and cultural factors that will be peculiar to the individual and impossible to model or to formalise in detail. Design decisions relating to these aspects are based on experience and intuition backed up by experimental evaluation. Formal models of interaction provide means for designers to incorporate these intuitions into system specifications that can ultimately be evaluated by implementation and experiment.

We have already observed that agent-based systems come in several forms. The model of an agent used by process algebras such as CCS, is of a process which communicates with other agents through fixed channels. The activity of the agents is synchronized by named events. On receiving an event an agent may replace itself with some other agent or set of agents and it suspends its activity pending the receipt of an event. The trace of an agent is the sequence of events in which it may participate and a typical use of agent-based specifications is to prove that a certain trace can (or cannot) occur.

The basic form of this model lacks means to express state and has been extended with state-based techniques to allow the specification of agents in terms of constraints on their state and their traces. Further extensions of the agent-based approach allow agents to access global state, as well as their own internal state and that of neighbouring agents in direct communication. It is possible to model various aspects of interactive systems in this way. For example, Abowd [1] proposes a hybrid approach based on work by Sufrin & He [140] which uses a combination of process-algebra with a model-based description of internal state.

A related model is the RED-PIE model [34], which considers sequences of user inputs (the **P**rogram) and an **I**nterpretation function which maps them onto states (or **E**ffects) of the system. The effect is then mapped onto outputs, the **R**esult and the **D**isplay. The result is the output in the wider sense of the observable state of the whole system, whereas the display is construed narrowly as the state of the display screen. The RED-PIE model permits reasoning about various forms of equivalence between different systems. It is primarily input oriented in that it seeks to define such equivalences over input traces.

Although these agent-based approaches do not model the cognitive and ergonomic factors that bear on user interaction, nonetheless they can be used to define properties of the link between system and interface that have a bearing on usability. These properties are defined formally in terms of two forms of correspondence (equivalence and indistinguishability) and a distinction between internal (state-based) and external (observational) descriptions of an agent. Some of these properties are set out below, to give a flavour of the model.

- **predictability:** externally equivalent agents are externally indistinguishable. This implies that observations of the (external) behaviour of the interface, past and present, is sufficient to predict its future behaviour, independently of the internal state of the agent.
- **determinism:** internally equivalent agents are internally indistinguishable. Whereas predictability relates to the observable behaviour of an agent, determinism is a property of its internal state and guarantees that knowledge of the current internal state is sufficient to predict the internal state for all time.
- **honesty:** externally equivalent agents are also internally equivalent. The observable (external) behaviour of an agent indicates its current internal state.
- **trustworthiness:** externally indistinguishable agents are also internally indistinguishable. Whereas honesty guarantees that an observer can immediately determine the internal state of an agent at any moment, trustworthiness only guarantees that an observer will eventually be able to determine the internal state for any particular moment.

This example shows one benefit of such formal approaches: they can enable the exhaustive enumeration of a set of possibilities. In this case, the two dichotomies gave a set of four possible combinations. Each of these corresponds to a property of the interface which might possibly affect usability. Not only does the formal approach show the relationship between these properties, it may even assist the researcher in discovering them. Another advantage of this kind of approach is that it provides a formal semantics for its components, so that resulting specifications are amenable to mathematical reasoning.

Formal approaches employing agents may also be used to characterize architectures such as PAC and MVC [1]. However, the ability to reason successfully about system implementations within such frameworks will depend on strict adherence to the formal models during system design and implementation. As has already been observed, neither of these architectures gives support for systematic development and it is left to the system designer to apportion responsibility for aspects of the interface to each of the components of the architecture.

There are really two distinct uses of agents in the context of UIMS. On the one hand there are the formal models, such as those derived from the RED-PIE model, which serve as system specifications or as *post hoc* descriptions of actual systems, and there are agent-like implementations, such as Sassafras [73], Tube [74] and

Rendezvous [75], which use an asynchronous event-response language to specify activity. However, such agent-like implementations do not offer support for applying constraints explicitly to the execution trace (i.e. the sequence of events in which an agent may participate), relying instead on the use of pre- and post-conditions to specify process-oriented properties of the system. Moreover, such implementations often use side effects, so that it becomes possible for the system implementer to use the systems in ways that do not conform to the agent model.

Agent-based specifications offer a means to structure specifications into modules and to reuse such specifications, but the reuse of specifications is not the same as the reuse of code. An open question is whether agent-based descriptions can be used to describe interface components in a way that enables the derivation of formal properties of interfaces built from those components. To achieve this it is necessary to have an operational model of the composition of interface components. An open problem is to define formal models of interface components, such that properties of a composite object may be derived from the properties of its individual components. Some work has been done in this direction in the context of logical input devices, which is discussed in 2.4.1. However, in general, it is necessary to rework proofs of properties when additional components are added to agent-based specifications [41].

Generally speaking, there is a trade-off between highly structured systems, which impose formal constraints on the implementation, and less structured systems which allow the interface developer more freedom, but at the expense of reusability. The separation of functional core and interface concerns implied in the Arch model is one example of this. The event-response systems referred to above do not enforce such a separation and therefore allow the construction of highly coupled systems in which the successful reuse of components depends on the ability of the developer to disentangle the dependencies between modules. One reason for this is that the communication and control mechanisms are required to be highly general, able to deal with functional-core-specific as well as presentation-specific entities. The set of objects employed for communication must be common to functional core and presentation and consequently likely to be composed of low-level entities, such as real numbers and strings. Descriptions of communications in terms of these entities are harder to understand and produce than descriptions at a higher, domain-specific level.

This is, of course, the point of the adaptor components in the arch model: to

allow the use of higher-level, domain-specific abstractions. Such adaptor components can be modelled as agents, but successful use of them requires that they be used consistently. In practical construction of interfaces, consistency may well be sacrificed to expediency: by-passing the adaptors can be simpler and more efficient in specific instances, even though such short cuts make the final product harder to maintain and reuse.

Agents have been used to model low-level interaction components, such as input devices and to model dialogue components. However, the process models on which they are based assume a tokenized stream of events. A general limitation of such models is that they have not been adapted to model visual properties of presentation components. Description of the visual interface is generally restricted to information content, rather than specifics of layout or topology. Thus an agent-based description can tell you that some piece of text will be displayed when a given action is performed, but details such as whether the text is upside-down, back-to-front or in an illegibly small font are not captured in the model.

2.2.5 Interaction toolkits

Interaction toolkits are one of the bases for the arch model, at the other extreme from the functional core. These are code libraries of interface objects such as buttons, menus, lists, text-entry fields and scroll bars, which can be used to implement the presentation layer. They may be based on a particular window system, though some toolkits offer cross-platform portability. Toolkits are frequently built using object-oriented languages so as to make use of class abstraction and inheritance for reuse and modification of components.

Some toolkits are just sets of routines in a standard programming language. Examples of this are MacApp (Object Pascal), which offers an object-oriented framework for implementing interfaces for the Apple Macintosh window system and CommonView (c++) which provides abstractions of toolkit functionality that are common to a variety of window systems (X, Macintosh and MSWindows). Other toolkits have their own special- purpose definition language, for example TCL, XDesigner and DevGuide.

Toolkits are frequently incorporated into interface builders, which offer the interface developer techniques for specifying the layout of components and limited access to the functional core through callback routines. Many interface builders employ direct manipulation techniques for the layout of components

and editing of their properties. For example, XDesigner allows the interface developer to see the assembled interface, while editing a tree-structure representing the hierarchy of components.

Interface builders are only suited to a small class of interface: they best suit those interfaces for which there is a predefined set of components whose layout can be determined at design time. They do not offer support for interfaces where the layout and properties of interface objects change dynamically.

Applications built directly using toolkits offer only a limited separation of the interface code from the rest of the application. The low-level details of input and presentation are hidden by the toolkit components (widgets), but structuring of dialogue is tightly coupled to the functional core, making modification and reuse more awkward.

2.2.6 Control and communication

UI architectures may be categorized according to the mechanisms used for control and communication. In any system which makes an architectural separation between the functional core and the interface it is necessary for there to be some way to coordinate the activities of the two components and to pass information between them. Control is the means by which dialogues are constructed and it concerns the temporal ordering of actions in the functional core and interface toolkit. Communication is concerned with how data is passed from one to the other. Three models of control are possible: internal, external and hybrid [68].

Internal control is when dialogue is implemented within the functional core. With internal control, the structuring of the dialogue and all aspects of the presentation are implicit in the functional core code. Consequently, functional core and interface are tightly coupled and modification of either the interface or the functional core on its own is difficult.

Multithreaded dialogues are hard to construct in systems which use internal control, since it is the responsibility of the functional core to implement mechanisms for dialogue control. The idea of factoring out of the functional core those mechanisms responsible for dialogue control and presentation is fundamental to ease of reuse and modification.

When the application is implemented as functional calls from within the interface, this is known as external control. Dialogue and presentation can be

specified explicitly in this case, using specialized tools and languages, but the interface component needs to retain information about aspects of the functional core semantics. In particular, the call-back routines will have to be parameterized with data structures appropriate to the functional core and this may require mechanisms for converting between the native representation of information in the interface component and the representation required by the functional core code.

Cockton has argued [25] that complete separation is only possible with the addition of a linkage component to mediate communication. Without this shared component, one or other of functional core and interface must exert control over the other. This idea was influential on the arch model, with its two adaptor components, whose purpose is to translate between the representations used in the interaction toolkit and those used in the functional core. Rather than implement protocols for converting information from one representation to another, it is possible to use a model in which data can be shared between functional core and interface. This shared data model gives a form of hybrid control in which functional core and interface execute concurrently, interacting with each other through the shared data. In effect, the data plays the role of the linkage component. This is a more flexible architecture, but to be useful the shared objects have to be chosen carefully, so as to support as wide a range of functional cores and interfaces as possible. For the purposes of reuse, it is necessary that interface and functional core can be changed without reworking of the definitions of the shared objects.

The Serpent UIMS is an example of a UIMS based on the shared data approach. It is described in its documentation as being based directly on Seeheim, employing presentation, dialogue and application layers [24]. In fact, though the terminology is the same, there are some differences in functionality between the three layers of Serpent and those of the Seeheim model.

The presentation layer is similar in function to that in Seeheim: it manages input and display through the use of toolkits. The dialogue component of Serpent performs functions of both the dialogue and application-interface component of Seeheim. It is responsible for translating between data formats used by the functional core and those used by the presentation. The role of the application-interface in Seeheim is taken by a shared database.

Dependencies between shared data can be expressed using a special purpose

language *Slang* and these are automatically kept consistent by the system.

The dialogue layer deals with low-level presentation aspects of the display, such as relative positioning of components. Integrating new input/output toolkits into the presentation layer requires the development of software to translate between objects in the dialogue layer and their corresponding toolkit components. These ad hoc routines fulfil the role of the presentation adaptor component in the arch model, though no specific support is provided for defining them.

2.3 Design environments

Existing tools for design tend to fall into one of two categories: dialogue tools for specifying sequencing and semantics of interface objects; and layout tools for specifying presentation details. The dialogue tools may be used to produce specifications which are implemented separately by a system constructor, or they may be fed into automated interface constructors. Layout tools are typically used for rapid prototyping and employ direct manipulation techniques to specify position and other visual properties of presentation objects.

Dialogue tools provide high-level abstractions and allow the designer to delay commitment to presentation specifics. This is often useful in the early stages of the design when conceptual models of the application are being produced and dialogue structured according to some kind of task model, formal or otherwise. However, at some later stage of the design it will become necessary to deal with the presentation details and there is a need for tools to incorporate these aspects later in the design. Some systems instantiate the presentation layer with default components, possibly using preference rules of some kind to select appropriate components. While this may relieve the designer of some programming effort, it also tends to restrict the final design.

Some automated tools which produce layouts from dialogue-level descriptions are intended as rapid prototyping systems in which the output is produced for early evaluation and reimplemented or hand tailored subsequently. Others intend the output to be final and achieve iterative refinement through interaction with the style rules used to generate the interface. The advantage of the latter approach is that it facilitates reuse of design effort and provides a record of design choices in the modified style rules. In contrast, hand tailoring of components requires greater programming expertise, leaves only an implicit record of abstract design choices and is less appropriate for reuse of components,

albeit it offers greater flexibility and control over the final product.

Layout tools force the designer into a premature commitment to presentation specifics. They generally make it fairly easy to modify designs, so it might be argued that the premature commitment is not a drawback. However, without means to record which choices are provisional and which permanent, there is a likelihood that premature commitments will become fossilized into the final design. Rapid prototyping systems for presentation layout are able to produce interfaces with a finished appearance in which there is no way to tell how much of the presentation is provisional. The ease with which interfaces can be constructed with such tools also leads to their abuse by neglect of the task requirements of the design in the mistaken belief that early evaluation and iterative refinement will lead to a usable product.

The problem of coordinating the activities of designers throughout the lifecycle of a system is considerably complicated by the variety of representations used at different stages of design. The model-based approach to design seeks to reduce this complexity by using a declarative model of the final system that can be amended and referenced throughout development. Once design is complete, the model may be executed by a standard run-time system. Reuse within this approach is achieved by building up catalogues of interaction components. It relies on the availability of suitable modelling constructs to express the final design.

The essential feature of the model-based approach is the inclusion within the design environment of metaknowledge about the application itself. The domain component and the user interface are generally modelled separately in systems which use this approach. The models define objects and actions that may be performed on those objects.

Examples of such systems are UIDE [141], ITS [57, 157] and HUMANOID [144,145,146]. The following sections consider these systems, with reference to model-based design.

2.3.1 UIDE

UIDE[1] is a model-based system, in which the application model assists design and analysis of the interface at design time and which is used to structure dialogue and provide intelligent help at run time [43]. Three layers of modelling

1. Note that here 'UIDE' refers to a particular system implemented by Foley and others, rather than to user interface development environments in general.

are used, corresponding to those of the Seeheim model: these are domain semantics, dialogue abstractions and interaction techniques.

Five basic entities are modelled by UIDE. (NB The designers of UIDE use the term 'application' in the sense of 'application domain'.)

- **Application objects:** domain-specific entities which are used as parameters for actions. These correspond to the domain objects of the arch model.
- **Application actions:** domain-specific actions which may be applied to application objects and which have pre- and post-conditions expressed in terms of the state of application objects. In the arch model, these are also domain objects.
- **Interface actions:** generic dialogues which capture actions found in various interfaces, but which are independent of any specific applications. These correspond to the conceptual objects of the arch model.
- **Interface objects:** objects which can be perceived directly by the user. The arch model refers to such entities as presentation objects.
- **Interaction techniques:** user-action sequences specified at the device level, e.g. keystrokes and mouse-clicks. In the arch model these would be interaction objects.

The application domain is modelled by subclassing application objects. Instances of application objects may be created at run-time, or pre-instantiated by the designer, if the number of objects is known in advance.

Application actions are characterized by pre- and post-conditions, parameters and constraints on those parameters. The conditions are expressed in terms of a restricted form of first order logic, without quantification. Application actions and their parameters are linked to interface actions, which are selected from a predefined set. Interface actions also have pre- and post-conditions which are used to derive dialogue structure automatically at run-time. Similarly, interaction techniques define sequences of input actions that are linked to interface actions.

The mapping between application and interface layers is one-to-many: a given application action may involve a sequence of interface actions. Similarly, the mapping from the interface layer to the interaction layer is one-to-many. Additionally, alternative mappings may be specified, so that a given interface action can be mapped onto one of several alternative interaction techniques.

At run-time, the dialogue manager checks the pre-conditions for application actions, enabling those whose pre-conditions have all been met and disabling

those whose pre-conditions are not satisfied. As each application action is enabled, the corresponding interface actions are enabled and their associated interface objects are made visible. By this means, dialogue sequencing is achieved automatically.

There is a certain overlap between interface and application concerns. For example, operations such as rotation or movement of visible objects are modelled as application actions, even though they have no direct significance to the domain semantics. Similarly, in order to model infix actions it is necessary to introduce a distinguished object, the currently selected object, in the application domain layer.

Kovacevic describes what he terms the 'compositional model' of user-interfaces. This extends research done with UIDE in order to allow structuring of the user interface into primitive components which can be reused to build new components.

> The compositional model is named after its central feature - representing a UI as a composition of primitives: it identifies elemental parts of UIs and basic structuring principles for composing them; furthermore, it defines how the composition depends on the application and the desired dialogue style. The model thus explains not only the structure and functioning of a UI, but also how the UI relates to the application it is part of and to different dialogue styles.
>
> Kovacevic [88]

The compositional model takes as its starting point the set of primitive components that are used in UIDE to define application semantics and extends them with rule-based reasoning to verify and enforce the relationship between application and interface. The model defines classes of component with specific roles in the interface.

The compositional model as implemented in UIDE and its variants has a number of limitations. Control of application semantics is mediated through explicit invocation of actions on named objects. There is a mapping from application objects and their attributes to presentation objects, but no inverse mapping. That is to say, the appearance of a presentation object may be changed by performing an application action, which alters the application state and thereby updates the presentation, but the converse is not possible: changes to the presentation cannot be mapped directly onto changes to application state. This means that interface actions have to be mapped explicitly onto application

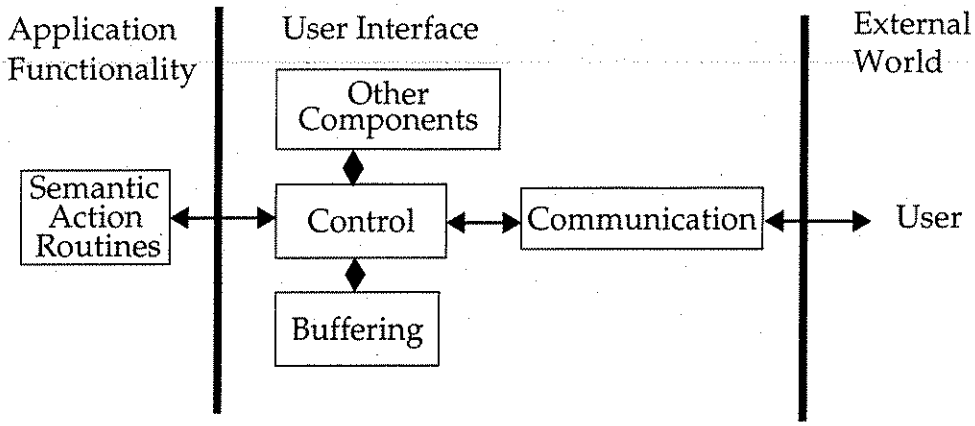


Figure 2.4 The essential classes of component and their interrelationship in the compositional model of the user interface, as defined by Kovacevic [88].

actions. The reuse of interaction techniques would be facilitated if it were possible to map directly between presentation state and application state, because in this case it would not be necessary for the UI developer to define the mapping between the technique and the action, in many cases it would suffice to specify a (bidirectional) mapping between application and presentation. In the UIDE family it is necessary to specify both the state mapping and the action mapping.

There are cases when a state-mapping alone is not sufficient. These are characterized by an underdetermination of the presentation. The presentation state does not correspond to a unique application state. In these cases, specification of the action mapping determines the application state. As an example of this, consider a simple drag-and-drop file-system interface, with a copy action. Suppose that three directories are represented and there are two identically named files (with different contents) in two of the directories. The action of copying either file to the third directory results in an identical presentation, but the underlying semantics is clearly different. Arguments such as these led to the formulation of the formal models based on agents which were referred to above.

In general, it seems that for effective reuse and ease of development, it is necessary to be able to specify both state-mappings and action-mappings.

In the application model, relationships between objects are modelled as compound objects. For example, the relation between a directory and its file contents would be modelled by specifying directory–file links as application objects. This is necessary in order to be able to define presentation properties for

the relation. Actions which effect the relation between the objects are then modelled as actions on the compound. Relations are therefore modelled *extensionally*, that is to say, by explicit enumeration. As a consequence, it is unwieldy to represent application domains in which there are overlapping hierarchies of objects, because of the large number of inter-object relations which must be represented [88]. In such cases, it would be more convenient to be able to define a relation *intensionally* as a predicate which is satisfied by the relation. Additionally, no mechanism exists for making inferences about relations, in order to derive new ones.

2.3.2 ITS

ITS [57, 157] is a system which allows the independent specification of interface style (otherwise known as 'look and feel'). It therefore provides means for separating aspects of the presentation. In terms of the arch model, it is concerned with the specification of presentation objects and the link between dialogue and presentation components. It provides means to link abstract descriptions of dialogue and presentation to standard toolkits.

ITS is based on a four-layer architecture:

- **The action layer:** corresponds to the functional core component; it consists of a code library of routines which access a shared data store. The data store itself mediates communication with the interface and may be considered as a set of domain-specific objects.
- **The dialogue layer:** is specified using frames, grouped into a hierarchical structure. The hierarchy represents an activation record of subdialogues. Attributes of frames allow the specification of actions, either named actions in the action layer, or activations and deactivations of other frames. Attributes additionally contain information about the nature of the data to be displayed, for example, the 'structure' attribute determines whether data in a field is continuous or discrete valued. The 'UserCan' attribute contains information on the actions which the user is permitted to perform on the data.The 'Purpose' attribute defines the kind of information that is to be presented, e.g. help, example, instructional. Information in the frame is used by the style rules to select appropriate interaction components. A dialogue manager controls the activation and deactivation of frames and control and communication between frames and the action layer.
- **The rule layer:** this uses conditions based on the attributes of the frame to select an appropriate interaction component for the dialogue. The ability to categorize frames is dependent upon the appropriate and consistent use of the attribute values by the interface designer. Some flexibility in style is provided by the ability to make rules condition-

al, both on the nature of the data to be displayed and on the position of the frame in the dialogue hierarchy. It is possible for a rule in a frame to override rules activated elsewhere in the hierarchy, so that descendants of the frame may be specialized. For example, a rule may override the default colour for title bars on windows, so that all descendants of the frame which activated the rule have the new default. The rule layer is a compile-time component, which is used to create a style program.

- **The program layer:** this is the run-time component produced by compilation of the style rules. Its role is to negotiate with the dialogue manager for the allocation of screen space. It is necessary to have a run-time component because of dynamic changes to the geometry and properties of the interface, such as occur when the number of components in a list changes, or a window is resized.

ITS provides a set of four basic categories of dialogue components, represented as frames which can be instantiated at run time. These are:

- **form-filling;**
- **choosing from a set of alternatives;**
- **list manipulation;**
- **reading text.**

The set of interaction components in ITS is extensible insofaras new components may be built up by composition of elements of the basic set. This set is itself extensible, though not without some programming effort.

ITS has been used to develop numerous applications, including an information system for EXPO '92 in Seville, and for a variety of rapid-prototyping applications. It is claimed [57] that productivity is greatly enhanced by the ability to separate responsibilities in interface development between system programmers (who write actions), application (domain) experts who construct dialogues, graphic designers responsible for style rules and toolkit programmers who construct the basic elements of interaction.

ITS is suitable for the construction of form-filling interfaces, where interaction is based on text entry and selection of actions by menus or other means: it is not a tool for constructing interfaces which require the manipulation of graphics, or which allow direct manipulation of graphical representations of data (excepting simple components such as thermometers). It must be said, however, that a great many applications are adequately served by the kind of interface that ITS can produce, and the ability to reuse styles is a step toward better usability by

providing standardization of interfaces. The style rules effectively encapsulate user interface guidelines, which previously have been subject to the interpretation of individual developers.

2.3.3 HUMANOID

HUMANOID [144,145,146] uses a template-based approach to the specification of the presentation component of the user interface. Templates provide hierarchical decompositions of *application objects* (i.e the domain objects of the arch model) into simpler components which are ultimately related to individual widgets (i.e the interaction objects of the arch model). HUMANOID is built upon Garnet and employs the constraint mechanism of that system to maintain data dependencies and perform automatic update of the display.

There are tools for describing five related aspects of the interface model. These are referred to as *application semantics, side effects, sequencing, manipulation* and *presentation.*

- **Application semantics:** the operations and actions provided by the application program.
- **Side effects:** the results of user interactions on the application semantics.
- **Sequencing:** the order in which user interactions may occur, achieved by enabling and disabling interactions.
- **Manipulation:** the operations available for interaction with presentation objects, at the level of user input (gestures).
- **Presentation:** the visual aspects of the interface, considered separately from their manipulation.

The run-time system of HUMANOID uses a shared data model in which domain objects are mapped onto presentation components using system defaults and through rules supplied by the interface designer. Reuse is facilitated through the separation of interface concerns into the five aspects listed above: descriptions of any of the five aspects form modules which may be reused in different interfaces. Rapid prototyping is possible because of the system defaults, so that the interface developer need not specify in detail all aspects of the interface.

HUMANOID also provides an interactive design environment in which the design of the interface is represented by a goal-structure. This enables the interface developer to keep track of design choices and records dependencies between the design solutions chosen. The design tool can also offer resolutions for design problems (through a set of defaults). It uses the application semantics

model to determine which actions have been made available to the end user. Pre- and post-conditions specified in the sequencing model are used by the design environment to determine which actions must be performed by the end-user and the interface developer may either select defaults offered by the system, or provide their own methods.

The HUMANOID system has much in common with UIDE and recently efforts have been made to integrate the two approaches into a system called MASTERMIND [111]. Whereas UIDE provides a richer set of primitives for modelling application semantics, it does not provide the same support for interactive development.

2.4 Interaction

There are a number of styles of interaction, ranging from text-based interfaces through to highly graphical ones. Considerable effort has been expended to find means to provide abstractions for interaction, ranging from formal descriptions of abstract interaction devices [35,40], through to concrete implementations of generic interaction toolkits [108].

In the following discussion the arch model will be used as a reference model and interaction styles will be described with reference to three levels derived from the model: the domain level, the dialogue level and the presentation level. Dialogue abstractions are actions in the interface corresponding to goals in the user's task structure, such as selection from a set of alternatives or setting a variable to a desired value. Presentation abstractions, on the other hand, describe the actions which may be used to manipulate objects in the interface. Examples are dragging or selecting objects and entering character strings. Whereas dialogue abstractions describe manipulations of objects understood in terms of functional core semantics, presentation abstractions are concerned with manipulations of visual properties of objects in the interface, treated as uninterpreted symbols.

One idiom that is commonplace in direct-manipulation interfaces is the object-action paradigm. In this idiom there is a distinguished domain object, the current selection, to which actions are applied. It is the equivalent of postfix notation in text-based languages in which an object identifier precedes the command name. The fact that the current selection exists only to support a given style of interaction implies a coupling between dialogue and domain levels. It has already been argued that this kind of dependency is generally undesirable, since it restricts the set of easily implemented interaction styles, although in this case it

must be said that the restriction is not too severe.

There may be several operations at the dialogue level, and many more at the interaction level corresponding to a single domain action. Systems such as UIDE and HUMANOID make these correspondences explicit. Dialogue abstractions can be realized in possibly many ways as sets of presentation abstractions and presentation abstractions can, in turn, be reduced to actions at the toolkit level. The relationship between objects and actions at each of the three levels gives one means to characterize the interface.

Correspondences between the different levels may be one-to-many, many-to-one and many-to-many. A single (conceptual) object in the application domain may be represented by one or many presentation objects and involve one or many functional core objects and actions. In practice, it is often the case that the relationship between actions in domain and dialogue levels is one-to-many. That is, a single domain action corresponds to a sequence of actions at the dialogue level.

Many interactions can be characterized as parameterized commands. The system UIDE, for example, employs this model exclusively, as has been described above. There is a great variety of dialogue styles available to set values for the parameters. These may be classified broadly as choice, analogue and text entry. With choice the user has to select from among a finite set of predefined options. Examples of choice dialogues are radio-buttons, menus and check-boxes. Analogue is where the chosen value is identified with some property of the presentation. Sliders and scroll-bars are examples of analogue devices. Text entry allows the user to specify values directly as strings. The command itself is selected by choice, say from a menu, or by gesture as when a drag-and-drop action has some meaning in terms of the domain semantics.

Dialogue boxes can be used to structure command parameters. The interaction consists of assembling a structured set of data which is then used to invoke some functional core functionality. They lend themselves readily to implementations by callback. Typically, once the user has signalled an intention to invoke the callback (e.g. by clicking on a button labelled **OK**) the dialogue box disappears, so there is no problem about maintaining consistency of functional core and presentation within the component.

A useful distinction is between direct manipulation of the application and

direct manipulation of the interface. The two differ in the way in which objects and actions are represented. Direct manipulation of the interface is basically an extension of command language interfaces in which commands and parameters are assembled and executed using interactive graphical dialogues to set values and choose options. With direct manipulation of the application, there is a continuous representation of application domain objects and application domain actions are represented at the dialogue level by distinct actions on the dialogue level objects.

The two styles of interaction may be distinguished by the way in which domain entities are represented (see Table 2.2).

Domain entity	DM of interface	DM of application
actions	selection	representation.
objects	selective valuation	continuous representation
properties	objects in part-whole hierarchies	properties of presentation objects

Table 2.2 Representation of domain entities

In direct manipulation of the interface, presentation objects and dialogues may be classified by function as

- **Selection:**
  Object selection: list boxes, icon arrays
  Command selection: menus, buttons, tool palettes
- **Identifiers:** text labels, icons
- **Information:** text, diagrams, help windows, alert boxes
- **View controllers:** scroll bars, window controls
- **Composites:** combinations of the above into dialogue boxes

Model-based user interface systems such as UIDE, HUMANOID, ITS and others have concentrated on the specification and generation of applications having direct manipulation of the interface. An aim of this thesis is to extend the approach by incorporating methods, based on constraint programming (see Chapter 3), to deal with direct manipulation of the application.

The interface as a whole may be characterized in terms of the number and kind of its components and their relation to domain entities. From the point of view of the designer, the simplest interface is one in which there is a fixed number

of presentation components, with fixed dependencies between them. In this case, the links between dialogue and domain can be expressed conveniently by extension (enumeration of all possibilities). Examples of this kind are form-filling interfaces and dialogue boxes. The most complicated interfaces are those in which the type and number of presentation components cannot be determined in advance, because they must be created in response to asynchronous events originating from the user, or the functional core, or possibly from other sources such as interrupts from peripheral devices. With this kind of interface, the links between functional core and dialogue components cannot be specified by extension, since they are not known in advance, and must instead be determined dynamically.

2.4.1 Abstractions for input

The input model used by all of the current graphics standards (e.g. PHIGS, GKS) derives from early work on interactive graphics [44,45,112,120]. It distinguishes different aspects of logical input devices, such as echo (visual feedback), measure (the computation of input values) and trigger (the moment at which the input value is to be measured). In the standard model [79], input devices may operate in one of three different modes. One of the key features of the model is the distinction between operating mode and the class of data input by the device.

- In **request** mode the application (functional core) blocks until the input device produces a value. Once the user has activated the trigger, the value will be computed and returned to the application. The device then becomes inactive.
- In **event** mode once the device is activated, values may be input repeatedly by the activation of the trigger. The values are queued, so that the application may deal with them in turn. Unlike request mode, the application does not necessarily block, because there may already be values in the queue.
- **Sample** mode is essentially the same as event mode, except that the user does not activate the trigger. The application can access the current value of the logical input device, without blocking.

The PHIGS model defines six classes of input data [5]: string; choice; locator; pick; stroke; valuator. The model has been criticized on a number of grounds, not least of which is the lack of formality in the description [133]. A number of attempts have been made to formalize the model and to augment it with additional modes: in particular a *selection* mode has been proposed to enable more than one input device to be active at the same time, with input coming from the selected device [35,

41], thereby enabling multithreaded dialogues.

The Garnet interface toolkit provides a set of concrete components, called interactors, which handle standard mouse and keyboard-based interactions in a platform-independent way [109]. The aim of interactors is to provide a separation of interaction techniques from specifics of presentation and feedback and of domain semantics. The underlying concept of interactors is that a small set of interaction techniques can be used to describe all direct manipulation interfaces. Similar considerations motivated the choice of abstract input devices in the graphics standards. Six distinct interactions are defined:

- **Menu:** control of actions through menus and buttons.
- **Move–Grow:** changes in size and position of objects.
- **New-Point:** creation of new objects.
- **Angle:** rotation of objects about a fixed point.
- **Text:** text entry. Facilities for text editing are limited in Garnet, however.
- **Trace:** produces a sequence of points from a pointing device, as a basis for gestural input.

As with the various other interaction abstractions, Garnet's interactors provide device independence, so that the interactions may be initiated with a variety of input devices, without the interface developer needing to be concerned with device-level aspects of the interaction, such as mouse-clicks or keyboard strokes.

There are strong parallels between the interaction model of Garnet and that of MVC, with interactors playing the role of the controller in MVC [108]. It was argued above in section 2.2.2 that one of the problems with MVC is the difficulty of assigning responsibilities consistently to the components of the triad. Garnet's interaction model alleviates this problem by providing a fixed set of components, specialized by parameterization, rather than subclassing. One of the consequences of this approach is that interactors require all presentation objects to respond to a standard set of methods for invoking behaviours such as feedback and for changing geometry. The visual behaviour of presentation objects is thereby separated from the interactor component. This contrasts with the interactors presented in the GKS and PHIGS graphics standards, where the form of feedback is encapsulated within the interactor and is not accessible from the functional core. The rationale for this is to give platform independence, since no guarantees can be made about the underlying capabilities of the hardware

platform. It should also be noted that, in contrast to the interactors defined in the graphics standards, which are abstract descriptions of implementations, Garnet's interactors are concrete implementations, usable only within the Garnet system.

Scheduling of the interactors is achieved through pre- and post-conditions. The semantics of the interaction may be customized by invoking actions before after and during interaction. It is necessary to have some means to handle both of these aspects of the interaction, but their inclusion complicates the model somewhat. The problem is that, whereas interactors allow generic abstractions of change to interface geometry, the link between interface geometry (presentation objects) and functional core is restricted. Therefore it is necessary to have individually-tailored routines for interpreting the effect of the interaction on the functional core in the particular context of the interface objects. Or, looked at another way, what is missing is a set of abstractions which map between presentation aspects of the interface and the functional core. In this thesis, we consider ways to provide such abstractions.

Card, Mackinlay & Robertson [18, 98] formalize user input in terms of physical operations using a vocabulary of primitive movements and a set of composition operators. Their analysis is based on the view that sequences of input may be considered as sentences of a formal language.

There are 8 primitive properties to describe input operations: linear position; linear movement; rotary position; rotary movement; force; torque; change in force; change in torque.

Three composition operators are defined:

- Merge composition: in which the input of the composite device is the composition of the inputs of the component devices, so that the composite accepts pairs of input, each element of the pair corresponding to one of the inputs for the individual components. The example of a mouse is used to illustrate this: mouse input may be considered as the merge composition of two linear input devices (such as sliders).
- Layout composition: describes the physical arrangement of input devices within a control panel or other composite device.
- Connect composition: is where the output of one device is connected to the input of another. For example, the mouse may be used to control a cursor, which in turn is used to manipulate a slider on the screen.

The devices so described may be physical, or virtual (implemented in software). By considering the *bandwidth* of each device (i.e. the fineness of control it can achieve),

the bandwidth of the composite may be derived and the suitability of the device for various tasks estimated.

Another approach related to the interpretation of physical movement is gestural input which attempts to provide some of the flexibility of working with pencil and paper. Typically, touch-screen and stylus are used for input devices [17,159]. Standard pointing devices (e.g. mouse) may also be used.

Recognition of hand-drawn symbols is not a trivial task computationally, and an important issue in gesture-driven systems is how to solve this problem [104,121]. Consequently, practical systems tend to employ a small set of easily distinguished symbols. The use of gesture to some extent eliminates the need for a palette of tools and thereby reduces errors associated with moded interaction. However, it also introduces a new difficulty for the user, because it becomes necessary to remember the set of symbols to be used. Kurtenbach and Buxton [91,92] discuss this issue and use the term *self-revealing* to refer to objects which provide continuous mnemonics to the user. Tool-palettes are self-revealing, whereas gestural input is not. Their solution to this problem is to provide mechanisms for the combination of gestural and orthodox input techniques.

From the point of view of this thesis, gestural inputs are interesting because they represent a mapping between graphical structure and semantics. However, we will not be concerned directly with the issues involved in gesture recognition and confine ourselves to the study of simple topological and spatial relations between objects. The mapping of spatial relations between presentation objects onto domain objects is considered in the next section.

So far in this section we have considered abstractions for concrete implementations: that is to say, concrete implementations of generic input behaviours (Garnet's interactors) and formal, or semi-formal, models of such generic input behaviours (the PHIGS model and its formalizations). The User Action Notation (UAN) represents a different approach, not so closely related to the concrete implementation, and more concerned with specification [69]. The UAN describes behaviour of users performing tasks with an interface. Tasks are represented as structures composed of subtasks, interrelated through a set of temporal operators, with the lowest level of abstraction being a set of user actions at the device level (mouse clicks and the like). It is primarily intended as a means of communication between interface designers and interface developers, although extensions have been proposed which can be executable as simulations

of user interaction (though not as prototypes of the interface) [59]. Hartson & Gray state the aims of the notation as follows.

> The need for communication among a multiplicity of cooperating roles in user interface development translates into the need for a common set of interface design representation techniques. The important difference between design of the interaction part of the interface and design of the interface software calls for representation techniques with a behavioural view - a view that focuses on user interaction rather than software.
>
> [67]

Three aspects of tasks are considered: user actions, system feedback and dialogue state (the latter referring to the state of both presentation and conceptual objects). The ability of the notation to express visual properties of the interface is somewhat limited. Presentation objects are assumed to have a context, which is the region occupied by the object on the screen. Input operations describing the use of a pointer may refer to this context, but it is not possible to describe how the contexts of objects interrelate, e.g. to specify that one context should be contained within another, or that they overlap. Other aspects of visual properties may be described as feedback (e.g. highlighting) or as dialogue state, but there are no constructs in the notation to formalize these in terms of a spatial or topological system, rather they have to be described as operations whose interpretation is left to the interface developer.

Extensions of the notation allow the representation of temporal aspects of tasks. Hartson & Gray describe operators for representing sequencing and interleaving of tasks, in the style of process algebras such as CCS & CSP [67]. Gray, England & McGowan describe an extension of the notation that allows the expression of real-time temporal constraints and also allows the parameterization of actions. The resulting specification can be executed to model user behaviour [59].

The UAN does not give executable specifications of interfaces: in fact it lacks the formal structure of a language, having no formal semantics and only a loosely-defined syntax. The syntactic structure which exists (e.g. the division into actions, feedback and state) is a convention, rather than a formal requirement and designers are free to use (or abuse) it as they wish. Nonetheless, UAN was primarily intended as an informal means of communication, and as such the lack of formal properties is not a drawback.

2.4.2 Visual aspects of the presentation

We now consider research which has addressed the problem of providing a link between the graphical properties of the presentation and the domain semantics. Constraint-based techniques have formed the basis for much of this work and there are now several UIMS which employ some form of constraint-resolution mechanism as a basis for linking attributes of domain and presentation objects. Indeed, HUMANOID, which was discussed above (section 2.3.3) is built on top of the constraint-based system Garnet. The characteristics and limitations of these systems depend in large measure on the nature of the constraint solution mechanism. We will consider constraint programming in detail in the next chapter and so defer consideration of the relationship between the underlying constraint system and the properties of the UIMS until then. In this section we give an overview of the systems and describe their characteristics and limitations without going into the implementation specifics whence they originate.

Gray, Waite & Draper describe a system *Iconographer* which provides a mapping between pictorial elements (icons) and functional core entities [58]. The mapping is between named attributes of the domain objects and named attributes of the individual elements. There is a simple direct-manipulation interface for establishing the mapping rules, based on the idea of a switchboard, in which individual attributes of classes of domain object are linked to attributes of the presentation objects. Values may be coerced to make them type compatible and some transformation of values (e.g. scaling) is possible through *object adaptors*. The mapping is one-way, in that there is no facility for mapping back to the functional core from the icon. Additionally, the form of the mapping is very simple, being one-to-one. In spite of this limitation, it demonstrates the advantage of declarative specifications of functional core–presentation mapping, and its designers claim that it facilitates rapid prototyping.

A number of systems have employed constraints to provide a two-way mapping between domain objects and presentation graphics. With such systems, the user has a set of tools available for making changes to the geometry of the presentation and there are rules which describe how the application data will change, in terms of the altered geometry. Garnet has already been cited as an example of this, in the context of interactors (see section 2.4.1). In the case of Garnet, the mapping is not truly bidirectional, because the interface developer must specify separate rules for mapping between presentation and domain

objects. The mapping may be specified in either direction (domain to presentation or presentation to domain), unlike Iconographer which only permits mappings from domain objects to presentation objects. However, a given attribute of the presentation will either be dependent on a domain attribute, or it will have a domain attribute dependent upon it: thus for a given domain–presentation object attribute pairing, changes can only be initiated on one side of the pair; changes made directly to the dependent attribute are not reflected in the independent one. Other constraint-based systems permit true bidirectionality, in which changes to domain or presentation attributes are reflected in either direction.

Kamada and Kawai describe methods for visualizing descriptions of abstract objects and relations between those objects [85]. They are not concerned specifically with user-interface issues and, like Iconographer, the mapping is from abstraction (domain objects) to presentation, but not vice versa. However, this work is relevant to this thesis, because it considers not only the representation of *properties* of domain objects, but also of *relations* between those objects. This makes their system TRIP (**TR**anslation **I**nto **P**ictures) considerably more expressive than Iconographer. It was observed above, in the discussion of UIDE (section 2.3.1), that in systems which do not provide explicit means to model relations between domain objects, the only way to represent them is by defining new classes of domain object, whose instances represent object tuples associated by the relation. However this approach to representing abstract relations is unintuitive, ontologically suspect and leads to considerable computational overheads as well as increasing the complexity of the interface designer's task. In TRIP, domain objects are represented by a relational structure modelling relations between those objects and a mapping is specified between domain objects and graphical objects and between domain relations and graphical relations.

Takahashi, Matsuoka & Yonezawa describe an extension of TRIP (TRIP2) which provides a framework for bidirectional mapping between domain and presentation [147]. They use four forms of representation, whose roles correspond to the four categories of object in the arch model.

- **Application data:** domain objects, representing data in the functional core.
- **Abstract structure representation:** this is a generic representation of the application domain, whose entities and relations are not specific to the functional core, corresponding to the conceptual objects of the arch model.
- **Visual structure representation:** the structure of the picture, described as a set of graphical relations among graphical objects; presentation objects in the arch model.
- **Pictorial representation:** the structure of the picture, in a form that can be rendered onto the display devices, and which is platform dependent. These are interaction objects in the arch model.

The structuring of the translation into intermediate representations allows reuse of the components by reworking the individual translations, as in the arch model.

The ability to represent abstract relations sets TRIP apart from most UIMS. However, there are a number of ways in which its ability to serve as a UI tool could be improved. The system as described in [147] is not incremental, so that any change to the visual or abstract structure results in recomputation of the whole scene and the authors report that there is work in progress to remedy this. The description of the mapping has to be made separately in either direction, though the authors also refer to work in progress to allow declarative specification of mapping rules that serve to describe the mapping between abstract and visual structure representations in either direction.

The set of visual relations which can be described in TRIP is limited to those that can be defined in terms of linear equations between points. This restricts the kind of visual relations which can be expressed. (The expressiveness of different constraint languages is considered in the next chapter.) Moreover, there are no constructs to define abstractions for dynamic properties, such as movement. In particular, TRIP does not present a continuous view of the domain objects, because it cannot deal with intermediate states of the representation formed when the user is in the process of creating a change. Consequently, it is necessary for the user to signal explicitly that the mapping between presentation and domain objects be updated.

Pineda considers qualitative approaches to describing the structure of visual representations in a constraint-based system called GRAFLOG [115,116,117]. This is somewhat different from the other approaches considered above, in that it does not attempt to define a semantics for the representation. However, we consider it

now because it provides an interesting abstraction for describing the visual aspects of presentation objects: users can refer to features of the representation based on the properties of those features. For example, the point which is formed by the intersection of two lines may be referenced as though it had been created explicitly as a separate object. This contrasts with other constraint-based graphic systems in which the only objects which may be referenced are those which have been created explicitly. The basic novelty of GRAFLOG is that it makes an important distinction between *accidental* features of the representation (those which have not been created explicitly as independent objects) and *necessary* features (which result from explicit constraints). The abstraction which this provides is extremely powerful: we will see in the next and subsequent chapters that constraint-based descriptions of visual representations which lack this kind of abstraction commit the interface developer to an unnecessarily detailed level of description. This is essentially the same problem that has already been mentioned in the context of UIDE (section 2.3.1), that of representing intensional objects, those whose existence is predicated on properties of other objects.

Mackinlay describes a tool, APT, for automating the presentation of information derived from relational databases [97]. This is based on a formalization of visual representations as compositions of elementary graphical languages. The work owes its theoretical foundations to Bertin [14]. Information to be presented is displayed using a composition of elementary visual idioms, such as mapping numeric values onto position along an axis. Three basic types of domain data are considered: nominal (unordered sets of items), ordinal (ordered tuples) and quantitative (based on numeric values). Selection of an appropriate presentation idiom for a relation is based on the structural properties of the relation (e.g. one-to-many, many-to-many) and the formal description of the graphical languages means that it is possible to prove that the chosen technique correctly encodes the given relation. In particular it is possible to show when a given technique gives an unambiguous visual representation.

Casner [19] describes a system BOZ that uses task-related information to select presentation techniques. Selection of presentation technique is guided by the nature of the information to be presented as in APT, but selection criteria also take account of how the presented information is employed in the performance of a task. Items of information that are necessary for the performance of a given task are grouped into visually coherent structures to emphasize their relatedness and

minimize the effort of visual search, using cues such as colour and spatial proximity to achieve that coherence.

Task descriptions are presented to the system using a task-description language, based on a set of *logical operators* that represent information-related actions in the task, such as comparison of values. The logical structure of the task is then transformed into a *perceptual structure*, using a set of rules for encoding logical operators as *perceptual operators*. These latter correspond to the primitive languages of APT.

Both APT and BOZ define only a one-way mapping (from domain objects to presentation objects). Their main contribution is to show how it is possible to achieve formalization of the nature of visual representations and to use this in order to present structural information about how domain objects are related. Moreover, the fact that the work has a formal grounding in logic means that the encoding of such information may be achieved automatically.

2.5 Conclusions

The previous sections have identified control and communication as an important issue in UIMS design, in particular

- how to ensure that communication between components is both flexible and sufficiently expressive;
- how to achieve that communication with the minimum use of resources without sacrificing completeness;
- how to organize control so as to achieve optimum efficiency in implementation.

The framework presented in this thesis uses a shared data (hybrid control) model of the user interface. The shared store models pictorial information. Both user and application can communicate with this store via agents. A full specification of this communication is the subject of later chapters. An important feature of this approach is its use of pictorial representations as the underlying means of representing application semantics.

For this approach to be usable, two things must be demonstrated:

- The representation allows a high enough level of abstraction to decouple sufficiently the user interface system from the functional core. If the communication between functional core and user interface is tied to low level details of presentation, such as absolute pixel coordinates, then reuse and modification will be unworkably complex.
- The representation can be implemented in a reasonably efficient manner. This in turn depends on the nature of communication with the store. In particular, whether the store is treated as a fully computational system in its own right, able to make deductions and answer complex queries, or whether it is seen merely as a repository or clearing house for data.

In contrast to techniques of specification which abstract away from the graphical details of the interface, in order to create formal specifications, the aim here is to retain an essential core of graphical information. In so doing, we may hope to gain certain advantages.

- It will be possible to argue formally about *visual* properties of the interface; current formal approaches allow reasoning about behavioural properties, but have little to say about visual properties.
- The specification will be simpler to use, employing *qualitative* descriptions of visual layout rather than requiring the designer to specify exact geometry.
- The ability to describe interface layouts and objects abstractly will facilitate their reuse.

Chapter 3 Constraint satisfaction problems

In this chapter we give an overview of constraint-satisfaction problems and constraint programming. The Introduction sets out the history and basic concepts of the subject. In section 3.2 we consider techniques for solution of constraint-satisfaction problems. The relation between constraint programming and logic is explained in section 3.3. In section 3.4 we consider solution techniques for qualitative constraint systems. The results from this section are important to the thesis as a whole, since the notation which is developed is based on the use of a system of qualitative constraints.

3.1 Introduction

The use of declarative techniques for the specification of interfaces is a natural means to reduce the load on the interface designer, particularly where abstractions are available so that the designer is not forced into a premature commitment to details of the design. Constraint-based specifications are declarative and have therefore attracted much interest in UIMS research. Systems have been implemented that employ constraint-satisfaction for specification, control and communication and graphical layout. Practical experience with such systems suggests that there are many compromises to be struck: these relate to efficiency of implementation, expressiveness, the ability to abstract from detail and the generality of the constraint domain. These will be examined in more detail below.

This chapter considers research work undertaken in different traditions, in particular artificial intelligence, logic programming, concurrent programming and UIMS practice. The aim is to identify approaches which will be applicable to the requirements of UI construction by placing practical work in UIMS within a theoretical context, showing its links to other research. We will consider the application of constraint-based techniques of programming and specification to the problems of user-interface design that were outlined in Chapter 2 and examine some of the benefits that this brings.

This introduction considers the basic concepts of constraint satisfaction. Section 2 describes mechanisms for constraint satisfaction in some detail. The next section considers the relation between logic programming and constraint languages. The overall theme of this chapter is that, whereas a great variety of applications and techniques exist for constraint programming, the range that has been applied to the construction of user interfaces is extremely restricted. We will examine the reasons for this and consider how other constraint-programming idioms may be used for UIMS. This will form a foundation for the subsequent chapters of the thesis.

We have already noted (see section 1.1) that the idea of interacting with graphical objects by applying constraints to their properties was first demonstrated in 1963 by Sutherland's Sketchpad system [143]. In its time, Sketchpad was a remarkable vision. Its hardware requirements were advanced for the time, requiring interactive graphics and considerable processing power. It allowed the user to manipulate graphical objects interactively on screen by applying constraints to their geometry. Its emphasis on facilitating communication between user and computer by providing concepts taken from a well-defined task domain is an early example of user-centred design: the system used constraints such as equal length, colinear points, parallel and perpendicular lines, which are basic concepts of Euclidean geometry, and essential to task domains such as technical drafting. The use of macros allowed the construction of abstractions to improve usability by extending the expressiveness of the system.

Sketchpad illustrates three ways in which constraint-based techniques can be valuable to the user interface designer:

- declarative specification;
- the ability to reason about and describe entities within a familiar task domain;
- abstraction away from low levels of detail.

Geometrical entities and spatial relations in Sketchpad were represented using real number coordinate-based equations for lines and curves. The solution techniques were based on iterative, numerical methods. Since Sketchpad, many different approaches to the solution of equations and inequations over $\mathbf{R}$ (the real numbers) have been investigated. Most of these defer the use of numerical techniques until the problem has been simplified as much as possible by rewriting (see section 3.2.4).

One very important area to which constraint techniques may be applied is spatial reasoning. This is such an important topic for the thesis that we will defer detailed consideration of the subject until Chapter 4. In this chapter we examine the theoretical basis of constraint solution techniques that are applicable to spatial reasoning.

3.1.1 Concepts of constraint satisfaction

Informally, the *constraint satisfaction problem* (CSP) is to find an assignment of values to a set of variables so as to *satisfy* a set of conditions (the constraints). Constraints are termed *unary* (resp.*binary, n-ary,*), when they refer to one (resp. two,many) variables. The *domain* of a variable is the set of values which that variable is allowed to take and may be defined explicitly as a unary constraint. A *solution* to a CSP is one in which the variable assignments satisfy all of the constraints.

In some variations of the constraint satisfaction problem all possible solutions are required, in others any solution is sufficient. When it is not required to satisfy all the constraints, then the problem is termed a *partial constraint satisfaction problem* (PCSP). In this case, the problem may be to optimise some global property of the solution, e.g. to maximize the number of satisfied constraints, or to minimize an error function over the value assignments.

When the domain of every variable is finite, then the problem is called the *finite domain constraint satisfaction problem* (FCSP). A naive approach to this problem is exhaustive search through the state-space of all variables. However, the general case of FCSP takes time exponential in the number of nodes. An active area of research is to find heuristics to guide the search, and ways to reduce the search-space of the problem.

There are various formulations of the constraint satisfaction problem in the literature. These are broadly equivalent, and the different statements of the problem reflect the wide variety of applications of constraint satisfaction techniques. We shall start by defining CSP, in which values drawn from a given set (the domain) are to be assigned to a set of variables, given that certain combinations of values are excluded.

In CSP, a constraint is a set of allowed valuations for a given set of variables with finite domains. Thus, for a set of variables

$V=\{v_1, ..., v_n\}$,

with finite domain

$D = \{d_1,...,d_m\}$,

a constraint, c, is a set of functions, c_i, from some subset V_c of V to D, such that for all c_i.

$V_c \subseteq V \wedge \text{dom } c_i = V_c \wedge \text{ran } c_i \subseteq D.$

In this formulation, each c_i corresponds to a valuation for the variables in V_c. Thus for a unary constraint, c, $|V_c| = 1$; for a binary constraint, $|V_c| = 2$; and so on.

A constraint satisfaction problem is a triple consisting of a set of constraints, C, a domain, D, and a set of variables, V.

CSP = <C,D,V>

A solution,s, to a CSP is a valuation function that is defined over all variables in V, such that for all constraints,c, in C, there is some c_i which is included in s.

$s: V \rightarrow D;$

$\forall c \in C; \exists c_i \in c \bullet c_i \subseteq s$

That is to say any set of variables for which a constraint has been defined has a valuation in the solution which is one of the valuations permitted for the constraint.

In FCSP the domain D is finite. In this case, obtaining a solution is easily understood as a form of search. In the worst case, a solution can be obtained by generating all possible valuations from V to D and checking them against the constraints. Some of the solution methods which improve on exhaustive search are described in a later section of this chapter.

We will sometimes refer to the *domain of a variable.* This should be taken to mean the set of candidate values for that variable. As information is added, we may be able to reduce the search space of the problem by eliminating candidate values which are known to be part of no solution.

3.1.2 Forms of solution

There are really three versions of CSP.

- **The decision problem:** here we are only concerned with deciding whether or not a given problem is soluble, without actually deriving a solution.
- **Finding a single solution:** in this case, any solution will suffice and incomplete methods may be applicable.
- **Finding all solutions:** here we wish to enumerate all solutions.

A problem is said to be *overconstrained* if no solution exists, and *underconstrained* if there are many solutions.

A *partial* constraint satisfaction problem is one in which partial solutions are allowed: a partial solution is a solution in which not all the constraints are satisfied. Often with such problems, there is an ordering on the partial solutions, and the problem is to find the best solution. A *constraint hierarchy* may be used in this context: it is a priority ordering of the constraints in a PCSP. The solutions may then be ordered by the priority of the highest priority unsatisfied constraint. Preferred solutions are those in which higher priority constraints are satisfied before lower ones. The lower priority constraints are effectively defaults which may be overridden.

Where the domain size of the variables is very large, or infinite, then methods of solution other than search are required. Techniques from equation solving have been applied to these problems, but the solution method must be chosen to suit the nature of the domain. For example, term rewriting can be used to solve linear and slightly non-linear equations over real numbers, but is not suitable for problems which involve general polynomials; for these, iterative methods may be employed.

Incremental solution techniques are those in which as constraints are added to, or removed from the problem, the new solutions are calculated incrementally, using the previous solution as a basis. This contrasts with methods of solution in which the whole problem has to be resolved whenever any change occurs. Incremental methods are valuable in UIMS for two reasons: they are faster, and response time is a critical factor in the user interface; and basing successive solutions on their predecessors may allow the user to anticipate the effects of any change.

3.1.3 Network representation of CSP

Any CSP may be represented as a network in which nodes represent variables and hyperedges[1] represent constraints. When the constraints of the CSP are binary, then the network consists of nodes connected by simple edges. This form of representation is often used to support graph-theoretic arguments about properties of CSP.

This form of representation is very important in this thesis, because we shall consider a formulation of CSP in which the domain values do not appear explicitly and the constraints are represented as labelled arcs in a network. We

1. A hyperedge is an edge which connects more than two nodes.

will use properties of the network to prove some important properties of the underlying constraint system.

3.2 A survey of techniques for constraint satisfaction

In this section, a number of algorithms for constraint satisfaction are discussed in detail, in order to consider their suitability for declarative specification of user interfaces.

Many solution techniques have been proposed for CSP and choice of an appropriate method is dependent on the nature of the specific instance of the problem. Particular factors which can influence the suitability of a solution technique are as follows.

- The state-space of the variables (their domain) which may be discrete or continuous, finite or infinite.
- The formulation of the constraints: for example, algebraic equations and inequations; logical clauses; explicit enumeration of allowed combinations of values.
- The topology of the constraint graph: the allowed arity of the constraints.
- The nature of the solution required: all possible solutions; any one solution; partial solutions (where it is not possible to satisfy all constraints).
- The existence of ad hoc techniques for the problem domain: arguments based on qualitative reasoning; heuristics that can be used to guide search for solution.

In this section we set out the main approaches to CSP and then conclude by considering how well each satisfies the requirements set out in Chapter 1.

3.2.1 Local propagation

One technique that has been employed for dealing with networks of constraints over very large or infinite domains, such as the real numbers, is to implement the constraint as a triple consisting of the following components:

- an *expression*, which evaluates true when the constraint is satisfied;
- a set of *methods* each of which creates a solution for some of the variables in the constraint (the outputs) when others (the inputs) are known;
- a *selection rule* that determines which method is applicable.

Typically, each method assigns a value to one of the output variables, and is used when all the other variables have known values. Solution of a constraint network is achieved by finding some method which can be activated (because its inputs are

known) and using it to augment the stock of known values.

This process is repeated until all variables have been assigned a value, or there are no more methods which can be activated, in which case the network is underconstrained. If the expression part of every constraint evaluates true, then the resulting valuation is a solution. If any expression evaluates false then the valuation is inconsistent with the constraints. Inconsistency occurs when two constraints attempt to assign different values to the same variable. In this case the constraints are said to conflict.

This technique is termed *local propagation*, since the known values of nodes in the constraint network are propagated to connected nodes by execution of the methods associated with each edge in the network. Where there is only one method the constraints are termed *one-way* or *dataflow* constraints. In this case, propagation is only possible in one direction. With *multiway* constraints, every one of the variables is assigned a value by at least one of the methods.

In all practical examples of local propagation, the selection rule is designed so that there is exactly one method for a given set of known values. Additionally the methods are such that, given a consistent valuation for the constraint, any method using a subset of the consistent valuation as inputs will produce output values identical to those in the given valuation. These provisos ensures that backtracking to try alternative methods is not required when an inconsistent valuation is encountered. The corollary of this is that the technique is only capable of providing a single solution to a constraint net.

Two variations on local propagation are *propagation of known values* and *propagation of degrees of freedom*. In the former case, the selection rule may make use of the actual values of the known variables. In the latter case, the selection rule is based on the set of known variables, without reference to their values.

The advantage of the former technique can be seen from the following simple example based on multiplication. The selection rule is shown in square brackets next to the corresponding method; assume that the selection criteria are checked in order so that only one method is selectable.

expression: A * B = C

methods:

(1) C := 0 [A = 0 ∨ B = 0]
(2) A := C / B [B ≠ 0 ∧ known(C)]
(3) B := C / A [A ≠ 0 ∧ known(C)]
(4) C := A * B[known(A) ∧ known(B)]

This is an example of a constraint using propagation of known values, since the selection rule makes use of the properties of multiplication by zero to propagate a value to C when only one input value is known. The corresponding constraint using propagation of degrees of freedom cannot make use of this information.

expression: A * B = C

methods:

(1) A := C / B [known(B) ∧ known(C)]
(2) B := C / A [known(A) ∧ known(C)]
(3) C := A * B[known(A) ∧ known(B)]

On the other hand, propagation of degrees of freedom has the major advantage that, because the selection rule does not make use of the variable valuation, it is possible to precompile solution plans based on the initial set of known variables. Such a plan consists of a sequence of methods. This does away with the need to search for selectable methods during solution. The resulting gains in execution time are significant. Practical examples of constraint-based UIMS almost all use some variation on propagation of degrees of freedom.

The use of known values means that local propagation cannot be used when there is only partial information about the variables. For example, local propagation can only handle inequations by converting them to equations so that

$A < B$

becomes

$A + \delta = B.$

The problem with this approach is that there will not usually be enough information to ground δ, so that no solution will be reached by propagation. Some of the other approaches that we shall consider would be able to return partial information concerning δ, such as a value-range or a symbolic expression.

Local propagation is often used in conjunction with constraint hierarchies. This technique is used when the problem is effectively underconstrained. Low-priority (*weak*) constraints provide default values for all the variables in the problem. These may then be overridden wherever necessary by high-priority (*strong*) constraints so that the solution never fails for lack of ground values.

Research on local propagation was carried out at MIT in the early 1980s [87,137,142]. Since then, variants of local propagation have been widely used in UIMS work (e.g. the systems Garnet [109], Thinglab [15], Rendezvous [114]). In essence local propagation reduces to the problem of finding orderings for the nodes in a directed graph.

Sannella [123] describes an incremental algorithm (SkyBlue) for the solution of *constraint-hierarchies*. These are essentially local propagation methods applied to a formulation of PCSP in which constraints are prioritized. The algorithms allow the incremental addition and removal of constraints and represent an improvement over a previous algorithm described by Freeman-Benson and others [48] (Delta-Blue). Whereas DeltaBlue could only handle one-way constraints in acyclic graphs, SkyBlue handles both cyclic graphs and multiway constraints, though at some cost in speed. The treatment of cyclic structures is to isolate them: the cycles may then be resolve by more powerful constraint solvers, or merely flagged.

Cyclic structures are generally a problem for methods based on local propagation and practical approaches to dealing with such structures range from isolating them (as with SkyBlue) to ignoring them (which is essentially the approach taken by Rendezvous). Whereas such *ad hoc* techniques are quite satisfactory from the point of view of achieving efficient implementations, they suffer from the disadvantage that the behaviour of any system which employs them is implementation specific and therefore it is difficult or impossible to derive any formal description of the behaviour.

3.2.2 Search heuristics

A different approach is needed when the underlying domain is discrete. Constraints over discrete domains specify allowed combinations of values for sets of variables and constraint satisfaction is the process of assigning values to the variables so that only allowed combinations are used. The fundamental algorithms for solution of constraint problems over finite domains were described in the mid 1970s by Mackworth, Montanari and others [100,106]. These were applied to areas such as machine vision and scheduling where many problems reduce to finding a consistent labelling for a graph. There is now a substantial body of literature on the subject.

When the constrained variables have finite, discrete domains, then it is possible to employ search as a means to solve the constraint satisfaction problem.

Unless the number of variables is small, then the state space of possible solutions will be too huge to employ naive search strategies. Various strategies have been employed to reduce the search space.

In the case where a complete solution to the constraints is required, wasted effort will occur during search when some inconsistent assignment to a subgraph of the constraint graph is held fixed while the remainder of the graph is searched. A related problem, known as *thrashing*, occurs when the search repeatedly generates the same inconsistent set of assignments to some subgraph.

The order in which variables are instantiated (have values assigned to them) and the order in which the values are tested can have marked effects on the time taken to find a solution. Several heuristics exist for ordering.

The *cycle-cutset* method finds a group of variables whose removal from the constraint graph would leave it tree-structured. That is to say, elimination of the cycle cutset removes all cycles in the constraint network. These variables are instantiated and then the remaining tree-structured network is solved. Tree-structured constraint networks can be solved without backtracking, by ensuring *arc-consistency* (see below). The search-space for the network is effectively reduced to that of the cutset. Therefore, best performance is achieved by minimizing the domain of the cutset. However, the problem of finding an optimal cutset is NP-hard, so heuristics are required to choose a cutset [33].

Another heuristic is to choose the variable which has the most constraints to instantiate first. Variable ordering techniques have been used widely in UIMS work, because they allow efficient solution when the same network has to be solved repeatedly, following minor changes. The instantiation order can be stored as a plan and reused [158, 48].

A number of incomplete solution methods have been proposed. (An incomplete method is one which does not guarantee to find a solution if one is present, but does guarantee that any solution found is genuine.) These have been widely used in the context of constraints posed as logic clauses. A surprisingly large number of CSPs may be framed as problems in logic: we will consider the relation between logic and constraint programming in section 3.3.

Improving on search may use methods based on look-ahead where the aim is to improve on the choice of next variable or value and look-back, which aims to improve on a point to which backtracking should occur.

Many of the methods based on search involve finding an ordering of the constraint graph which can be instantiated without backtracking. Freuder [52,53] describes conditions for backtrack-free search, which are based on consistency properties (consistency is described in section 3.2.3). These methods relate the width of the constraint graph to the degree of consistency between the constraints. We consider some of these methods below when we have defined consistency.

3.2.3 Consistency

The search space can be reduced through enforcing various degrees of *consistency*. This is achieved by maintaining a set of candidate values for each of the variables and eliminating those which are known not to be part of any solution. Various degrees of consistency have been used to simplify constraint satisfaction problems.

Node consistency is obtained by the elimination of all values that are explicitly excluded by unary constraints on the variables. In the following simple example based on font selection, it is required to assign fonts for menus, buttons and text labels from the set **A**vant Garde, **B**ookman and **C**ourier, subject to the constraint that each component should have a different font and that not every font is allowed for every component. Three unary constraints represent the allowed values.

menu $\in$ {B,C}

button $\in$ {C}

text $\in$ {A,B,C}

Initially all three variables, menu, button and text have the candidate values {A,B,C} in their domains, but clearly it is pointless to check valuations which contain a value that has been forbidden explicitly by a unary constraint. Node consistency is achieved by removing all such forbidden values from the candidate sets. In the example, it is necessary to remove the value A from the domain of menu and to re-

move A and B from the domain of button.

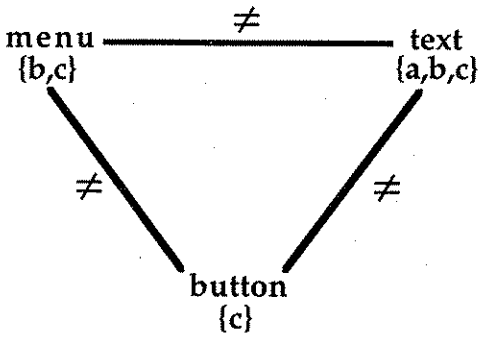


Figure 3.1 A simple constraint network

Arc consistency is defined with respect to pairs of variables: it ensures that for all values in the domain of any given variable there is at least one allowed value in the domain of its pair. A given set of variables is arc consistent if all pairings of its elements are arc consistent. In the example, three binary constraints enforce the condition that each object have a different font

menu ≠ button

button ≠ text

text ≠ menu

The domain of menu is already arc-consistent with that of text, because both values in its domain are consistent with the value a for text. Similarly, the domain of text is arc-consistent with that of menu, since

for value a of text menu can be assigned b or c

for value b of text menu can be assigned c

for value c of text menu can be assigned b

Also, the domain of button is arc consistent with both menu and text, but note that the relationship is not reflexive, since neither menu nor text is arc-consistent with button, since both of them contain values for which no consistent assignment in button exists. For example, if menu takes the value c, then there is no allowed assignment for button. Therefore, to achieve arc-consistency, it will be necessary to restrict the domains of menu and text by removing the value c from both. However, in so doing, we will introduce a new inconsistency between text and menu, since the assignment of the value b to text will no longer have any consistent assignment in menu, thus it will be necessary to restrict the domain of text by removing the value

b.

In this case, enforcing arc-consistency results in the domains of the variables having only one value each, so that there is only one solution possible, which may be found without search. This is not generally the case. In fact, enforcing arc consistency does not even guarantee that a solution can be found. For example, the network in the diagram below is arc-consistent, but has no solution.

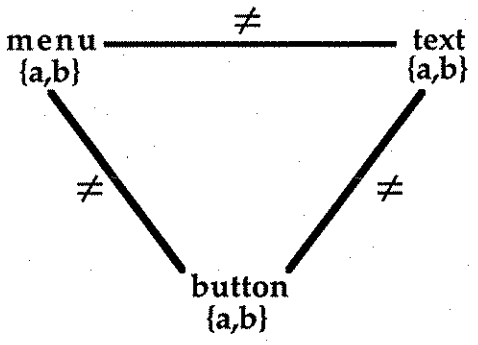


Figure 3.2 An arc-consistent network with no solutions

Arc consistency has been used for solution of systems involving only binary and unary constraints and fast parallel solutions have been proposed [28]. The basic algorithms for establishing arc consistency (which he terms AC-1,2,3) were described by Mackworth [100], who also describes algorithms for the more general notion of *path consistency.* This work was based on theoretical foundations established by Montanari [106].A network is path consistent if, for any two variables in the network, their domains contain only those values which are permitted by all paths connecting those variables. This is clearly a stronger condition than arc consistency, although it does not guarantee that a solution can be found. Improved algorithms for arc and path consistency were described by Mohr & Henderson [105] and Han & Lee [63].

The more general notion of *k*-consistency ensures that for any group of *k* variables, for every value in the domain of every variable there is at least one allowed assignment to the other *k-1* variables. If the degree of consistency of a constraint network is sufficient, then the network can be solved without resorting to backtracking. The required degree of consistency depends on the *width* of the constraint graph, a graph-theoretical property which is related to the degree of interconnection of the nodes. Thus, in theory at least, it is sufficient to enforce *k*-

consistency to the required degree in order to enumerate all solutions of the problem. Unfortunately, methods of enforcing *k*-consistency are often more expensive than direct solution of the underlying constraint satisfaction problem.

The lower orders of consistency are cheaper to obtain: 1-consistency corresponds to node-consistency and 2-consistency to arc consistency. It has also been shown that path consistency is equivalent to 3-consistency. This follows from a result established by Montanari that if every path of length 2 is path consistent, then so is the whole network [106]. Freuder describes algorithms for enforcing *k*-consistency [51] and also shows how the efficiency of solution is related to graph-theoretical properties of the constraint network [52,53].

Dechter and Pearl have considered the special case where the width of the graph is 1, in which case it is tree-structured and present an algorithm for solution without backtracking which involves imposing arc-consistency (essentially a form of look-ahead). The case in which the graph has width 2 may be solved by imposing path-consistency, although this has the problem that it may add arcs to the constraint graph, thereby increasing its width. Dechter and Pearl therefore propose a modified version of consistency, sufficient to achieve backtrack-free search, but which is more efficient and results in fewer additions to the constraint graph. The form of consistency which they propose is directed, so that an arc is *d-consistent* if all values in one domain have support in the other, but not necessarily vice versa. They claim (erroneously) that a two-pass application of the algorithm for enforcing d-consistency produces full arc-consistency in trees in $O(nk^2)$ time (n variables, with k values in the domain). In fact this is not so, since the reverse pass may remove values which supported the forward pass. Mackworth's algorithm AC-3 gives an $O(nk^3)$ time for tree-structured graphs.

A problem with the use of techniques based on consistency is that local changes to the domain of a variable may propagate inconsistencies globally. This can make the incremental update of the constraint set a computationally expensive process.

We have observed already that there are strong parallels between methods based on search and methods for local propagation. Particularly in the case of search methods which involve reorderings of the constraint graph. However, the consistency methods we have reviewed in this section have never been applied to the field of UIMS construction. The reason for this is that they are essentially methods based on finite domains and the most important domain for UIMS work

is infinite ($\mathbf{R}^2$). However, we will describe in section 3.4 a modified formulation of path-consistency which can be used to express relations between variables, without specifying their values explicitly. We will argue that this approach, though novel in UIMS work, may be the basis for the form of description which we outlined in Chapter 1.

3.2.4 Closed-form solutions and term rewriting

Closed-form solutions are those in which it is possible to arrive at a solution through application of a predetermined set of methods. Dataflow and other forms of local propagation fall into this category, as used in Garnet [109], and the technique can be extended to include multiway constraints when preceded by formulation of a plan (evaluation order) through search of the dependencies implicit in the constraint graph.

The solution of problems involving real numbers would appear at first sight to preclude the use of the search techniques employed for finite-domain problems. In fact this is not so in the case where closed-form solutions exist. In such cases the problem can be transformed to a consistent labelling problem, where the labels correspond to methods in the constraints. Once a consistent labelling has been achieved, then the real-valued solution to the original problem is obtained by applying the methods corresponding to the labels.

Kramer [89] has shown that a usefully large set of geometric constraint problems can be solved by a technique known as *degrees of freedom analysis*, which yields numerically stable, robust polynomial time solutions. In this method, constraints are expressed in terms of kinematic restrictions on objects, such as the constraint that an object may rotate about a fixed axis only, or that a point must lie upon a line. The basis of the method is to solve the constraints by constructing an action sequence of elementary geometric transformations, subject to the constraint (a *metaconstraint*) that no action may change a quantity which no longer has any degrees of freedom.

Kramer's method has some similarities with augmented term rewriting [94], which is described below. The solution of kinematic problems by degrees of freedom analysis is confluent. That is to say, all orderings of an action sequence that solves a given set of constraints will yield identical solutions. One virtue of this from the point of view of HCI is that the behaviour of the system does not become unpredictable as a result of non-determinism in the solution ordering. Such unpredictable behaviour is a problem with numerical methods of solution,

where small changes in the constraint network can lead to drastically different solutions. Some authors invoke a *principle of least surprise* to choose between competing solutions. That is to say, incremental changes to the constraint network should be solved in a way which minimizes changes to the variable assignments and, by implication, the presentation of the interface.

Term rewriting (also called *equational programming*) is a natural approach to the solution of simultaneous equations (and hence constraints) insofar as it resembles the pencil and paper solution techniques for such systems. In this technique, constraints are manipulated symbolically by applying reductions to expressions (*redexes*) until an irreducible form is reached. This is known as the *normal form*. In systems which use term rewriting, it is usual to impose syntactic restrictions on the allowed forms of redex, so as to ensure *confluence*. That is to say, a system of redexes will always reduce to the same normal form, irrespective of the order in which the reductions are applied. Confluence is a convenient property since it reinforces the non-procedural nature of constraint programming by ensuring that the order of declaration of constraints does not affect their meaning.

As with logic programming languages, such as Prolog, symbolic results in the form of equations may be the outcome of a reduction. Unlike logic programming languages, term rewriting systems do not have the inbuilt ability to generate multiple solutions to a problem. This is a consequence of the syntactic restrictions which guarantee confluence.

Leler has implemented a general purpose language called Bertrand [94] in which he extends term rewriting with variable binding and the ability to define new objects and types, calling the extension augmented term-rewriting, and shows how it may be used to define and solve systems of constraints.

Term rewriting has the advantage of allowing the programmer to express redundant constraints in order to improve the efficiency of solution. In fact there are certain forms of equation that cannot be solved by local propagation techniques, which are nonetheless amenable to term-rewriting. In particular, slightly non-linear equations. This advantage is a result of being able to modify the topology of the constraint network. In fact, term rewriting is a special case of graph-rewriting.

In the Equate system Wilk [158] uses term-rewriting in the context of object oriented programming in order to reduce constraint expressions to message

sends. This is a significant piece of work because it shows how constraint programming may be implemented in an object-oriented environment without violating encapsulation. Much work involving constraints in UIMS has been based on object-oriented platforms. The motivation for this is the availability of interface components within such platforms. Constraints in such systems often directly change the values of instance variables within interface objects. This violates encapsulation, the principle of data hiding which is widely cited as a benefit of object-oriented programming.

Equate distinguishes between read and written variables, constructing an acyclic dependency graph of instructions as a solution. Expressions ultimately reduce to instructions (method sends), via term-rewriting. As any expression may have one or more reductions, there will generally be a set of possible solutions to any given constraint expression. Equate does not choose between available solutions, but instead leaves control of implementing the solution to the application. It is possible for solutions to block during execution (this is an inevitable consequence of allowing conditional constraints), in which case it is the responsibility of the application whether to backtrack before choosing an alternative solution from the current solution-set.

Another approach to the problem of encapsulation is described by Wadler [154] in his work on *Views*. The motivation for this work was to allow equational reasoning at the same time as data abstraction. Encapsulation in object-oriented languages is a form of data abstraction. In this approach, rather than hide the implementation of a data type, multiple views of the same type are made public and invertible functions map between the various representations.

A more general form of rewriting is expressed using graph grammars. In these, the objects which are manipulated are graphs consisting of nodes and connecting arcs. The rewrite rules consist of three parts: an embedding, which is used to match some subgraph of the graph under consideration; a set of additions of arcs and nodes; and a set of deletions of arcs and nodes. Additions and deletions are performed using name bindings established by the embedding. Graph grammars have been used for data modelling and query construction in databases [Gyssens, Paredaens & van Gucht 1990] and for the specification of specialized document editors [Schurr 1989]. Graphs have been widely used for the modelling of knowledge, since the introduction of semantic nets [134]. Graph grammars augment these models by using production rules to specify dynamic

behaviour.

3.2.5 Infinite domains

Numerical and geometric problems fall into this category. Solution techniques may be divided into two categories: those which use a class of iterative numerical methods known as relaxation to obtain an approximate solution and those which rely on the existence of closed-form solutions to particular subclasses of problems.

Many geometric problems can be formulated as a set of simultaneous polynomial equations. Constraint-based descriptions of graphical layout frequently give rise to problems involving the solution of simultaneous polynomial equations over a set of real-valued variables. A simple example is the constraint that the distance between two points be fixed.

$$(x_1 - x_2)^2 + (y_1 - y_2)^2 = d^2$$

This class of constraint problem is therefore very important for GUI work. Several systems have employed relaxation to solve such constraints, notably Sketchpad [143] and Thinglab [15]. Experience has shown that, although this approach may be used to solve a wide variety of problems, it is computationally very expensive and can lead to unsatisfactory performance where real-time feedback is required. Those systems which use numerical relaxation generally do so as a last resort when other solution methods have been used to reduce the size of the constraint network.

3.3 Constraint programming and logic

Several authors have explored the relationship between the constraint satisfaction problem and logic programming. Colmerauer [27] has incorporated constraint satisfaction mechanisms into Prolog in the language Prolog III. Jaffar and Lassez [81] propose a family of constraint logic programming languages united by a common syntax and operational semantics, which they term the CLP() framework. Mackworth [101] attempts to unify various approaches to constraint satisfaction by viewing the constraint satisfaction problem as a restricted logical calculus. Saraswat [124,125,126] defines an operational semantics for concurrent constraint programming that subsumes actors [2], agents [103], and logic programming.

Mackworth shows how constraint problems may be converted into logic problems by creating a predicate for each variable in the constraint network and then defining the domains of each by *extension* (i.e. by enumerating all possible values). The problem itself can then be posed as a query, and the solution(s) to the

problem are found as instantiations of the variables in the query. Thus, in Prolog, the simple example used above on font-selection could be defined as follows.

First define a predicate for each variable in the constraint satisfaction problem and define its domain by extension.

```
menu(b). menu(c).
button(c).
text(a). text(b). text(c).
```

Now do some necessary 'housekeeping'.

```
notEqual(a,b). notEqual(b, a).
notEqual(b,c). notEqual(c,b).
notEqual(a,c). notEqual(c,a).
```

Finally, pose the constraint-satisfaction problem itself as a conjunctive query:

```
?- menu(A), button(B), text(C), notEqual(A,B), notEqual(B,C).
A = a.
B = b.
C = c
```

The solution is the instantiation of the variables A, B, C.

Declaration of constraints is a natural extension of logic programming and there is no need explicitly to program the search for a solution, because the resolution method of the programming language will perform the search. However, logic programming languages can be very inefficient for solving large constraint networks, unless some means to control search is provided. Moreover, the requirement to define values by extension is awkward and counterintuitive.

3.3.1 The CLP() framework

Solution by unification, as used in standard Prolog, cannot resolve constraints posed in the form of simultaneous equations. This kind of problem occurs frequently in the description of graphical layouts. Some work has been done to extend logic programming to cope with this kind of constraint. Two approaches have been used. In one due to Jaffar & Lassez [81], the unification mechanism which underlies Prolog is replaced by a constraint satisfaction mechanism. In the other, due to Colmerauer [27], unification is supplemented by domain-specific constraint satisfaction mechanisms. Jaffar and Lassez's approach gives rise to a family of constraint-programming languages, CLP. The CLP() scheme [81,82] is a framework for defining programming languages. An instantiation of the scheme,

CLP(D), is a particular programming language and is achieved by specifying an application domain, D, and a set of functions and relations on the objects in that domain. The framework provides the operational semantics and syntax of the language: in order to create an instantiation, it is necessary to provide a constraint solver for the given domain.

A typical instantiation of CLP is the language CLP(R), where R is a structure combining trees and real numbers [82]. In this language, unification is effected by equality constraints between trees. Implementation of the language requires the provision of a set of constraint satisfaction mechanisms for the structure R. The constraints which can be solved in CLP are linear equalities and inequalities over the real numbers. Nonlinear equations can be handled by the system, provided they become linear following the grounding of some of their terms.

The advantages of implementing specialized constraint languages are twofold.

- They allow the designer to work with familiar concepts, at an appropriate level of abstraction.
- They permit the use of efficient techniques of implementation.

The following observation of Jaffar et al regarding CLP(R) emphasizes these advantages. It refers to the contrast between working in CLP(R), where arithmetic constraints can be declared without explicit coding and working directly in logic, where it is necessary to provide explicit code for dealing with arithmetic objects.

> ... CLP(R) facilitates declarative programming because of enhanced intuition when reasoning about programs. More specifically, working directly in the intended domain of discourse contrasts with working in an Herbrand Universe, wherein every semantic object has to be coded into an Herbrand term. This coding enforces reasoning at a more primitive level. In CLP(R), on the other hand, local properties of complex problems can be stated naturally as arithmetic constraints, and then the problem as a whole can be represented by the use of rules. [82, p340]

Similarly, for reasoning about pictures, the domain of real numbers is overly low-level and inhibits the expression of properties of the picture in a natural way.

The implementation of CLP(R) is based on a hybrid constraint satisfaction mechanism, in which a number of solution methods are employed. Some of these methods, such as the inequality solver which uses a modified Simplex method, are computationally expensive, although they permit the expression of a wider

range of constraints. In such hybrid systems, where they are used to solve problems that do not require the full generality of the system, then the costs involved should be less than when the full generality is involved. Whereas the provision of very powerful solution mechanisms allows the expression of a wider range of constraints, it is often advantageous to restrict the set of allowed constraint expressions, so as to make it impossible to define intractable problems.

The ability to apply constraints dynamically is essential if they are to be used to program the behavioural aspects of the user interface [164]. A useful distinction can be made between static and dynamic constraints: the former are rules applied directly to specific variables, whereas the latter are templates that can be instantiated at run-time. Dynamic constraints offer a natural means to express control mechanisms, in which the current set of constraints is used to determine how future constraints are added. This makes programming idioms such as recursion possible. Where constraints can be applied dynamically, considerations of efficiency require the use of incremental solution mechanisms.

The versions of Prolog (II & III) presented by Colmerauer [27] may be considered as examples of CLP() languages, although they are not explicitly described as such. In these, constraints are special predicates over Booleans, rational numbers and infinite trees. However, the implementation of Prolog III and CLP(R) differ in that the latter implicitly employs methods that have to be invoked explicitly in Prolog II/III. For example, in CLP(R) slightly non-linear constraints (those of the form $X_0 = c*X_1*X_2$) are automatically delayed until enough information is available for their solution, whereupon they are 'woken up' and passed to the constraint solver. On the other hand, in Prolog II, constraints involving multiplication of rationals are not supported directly by the system, but the effect of the 'wake-up' mechanism of CLP(R) can be programmed explicitly using the **freeze** construct, which delays evaluation of a constraint until all variables are ground.

In CLP(R), the basic constraints

$t_1 = t_2$, $t_1 < t_2$, $t_1 \leq t_2$,

where t_1, t_2 are arithmetic terms, and

$t_1 = t_2$,

where t_1 and t_2 are non-arithmetic, are called *primitive*. Atoms are predicates of the form

$p(t_1,t_2,...,t_n)$

and rules are

$A_0 :- a_1, a_2, a_3, ..., a_k,$

where the a_i are either atoms or primitive constraints, A_0 is called the *head* and the a_i are called the *body*. Goals have the same form as the body of a rule. Computation proceeds by a sequence of derivation steps in which a goal G2 can be derived from a goal G1 either by solving one of the constraints in G1 and augmenting G1 with the solution, keeping all other atoms and solved constraints unchanged, or by replacing one of the atoms of G1 with the body of a matching rule. A derivation is a sequence of derivation steps and if successful will contain only solved constraints.

The necessity for the use of an incremental solver is clearly seen from the model of computation, in which successive derivation steps add the solution of a single constraint to those already solved.

3.3.2 The cc() framework and concurrency

In the same way that CLP() is a generalization of logic programming, so cc() generalizes CLP() by the inclusion of operators that enable synchronization of concurrently executing agents.

The ability to express temporal dependency is fundamental to dialogue definition. Multithreaded dialogues require operators that denote synchronization and parallelism; deferred evaluation may add to the expressiveness of the dialogue description. The ability to compose dialogues retrospectively, while not strictly essential, is clearly an asset. The agent model of computation, as described in Milner's CCS [103], offers such an ability, as do actors [2]. Both models support composition, but differ in that the composition of agents is itself an agent, whereas the composition of actors is not. Moreover, the CCS model uses synchronous communication between agents, whereas communication between actors is asynchronous.

The research into finding a unified semantics for logic and constraint programming has been paralleled and, to some extent complemented, by work on concurrent constraint programming. Saraswat proposed a reference language CP, which includes Prolog, concurrent Prolog and pure Horn clause programming as special cases [124]. This introduces a set of operators that can be used to express temporal relationships which can guide search for a solution. He extended these ideas to cover concurrent constraint programming in the cc() family of languages [124,125,126].

The ideas embodied in cc() form a powerful set of tools for the language designer. Firstly, they define a semantics for the

> ... intuitive pre-theoretic visions about computing with constraints, evidenced in some of the work of such people as Sutherland, Sussman, Steele, Borning and some of their colleagues.
>
> [126]

Secondly, they define mechanisms for communication that are applicable to distributed systems and naturally support concurrency. In cc() languages, communication is achieved via a store of constraints. Concurrently executing agents communicate with each other via the constraint store, using two basic operations: these are *tell* and *ask*. The tell operation augments the store with a new constraint, provided that the constraint is consistent with those already in the store, and the ask operation blocks until a constraint is entailed by the store. This view of computing with constraints contrast with the traditional view of computing in which there is a store of variables, each of which has a definite, known value. These languages support reasoning using *partial information* in which evaluation of a constraint may be deferred until the store contains sufficient information either to entail the constraint or its negation: this is the blocking ask operator. Use of the blocking ask construct can be used to build local propagation systems, such as those described by Steele [137].

The cc() family is closely related to, and inspired by, Milner's CCS. However, whereas in CCS synchronization is achieved through directly visible actions, in cc() it is through the use of ask and tell on a shared store of constraints. A notable feature of cc() is that it employs a *committed choice* strategy whereby successive tells are only permitted if they do not violate the existing constraint store. The implication of this is that once a constraint has been successfully told, it is guaranteed that it can be told again, also once a constraint has been successfully asked it will always be successfully asked. This property of stability enables certain implementation efficiencies to be achieved.

Kahn and Saraswat [83] have shown that both agent-based languages and actor languages may be given a semantics based on concurrent constraint programming, i.e., within the cc() framework. They have implemented languages *Janus* and *Lucy* to explore these concepts. Lucy conforms strictly to the actor model of communication in which communication is mediated by message queues known as mailboxes, one per process. Janus provides a more general

language in which multiple channels of communication per process are allowed. This corresponds to the agent model of computation. Permitted constraints in these languages are equality constraints for the tell operation and simple tests such as arithmetic relations and type tests for the ask operation.

CLP() and cc() languages differ in that the latter support the notion of *constraint entailment* which is the basis for the ask construct. These languages provide mechanisms for checking whether or not the store of constraints entails a given constraint. This will be the case when all states allowed by the store are also allowed by the constraint. Entailment is the basis for two powerful operators: the cardinality operator and the implication operator. The general syntax of the cardinality operator is

$\#(l,u,[c_1, \ldots c_n])$

where l and u are lower and upper bounds and the c_i are constraints. The declarative semantics for this constraint is that at least l and at most u of the c_n will be satisfied. A 'don't care' value can be used for l or u.

The implication operator has the form

c -> E

where c is a constraint and E is an expression in the language (either a constraint or a functor expression). The operational semantics of the implication operator is that if c is entailed by the constraints in the store, then the statement is equivalent to E, whereas if ~c is entailed by the store (i.e. c is known to be unsatisfiable) then the expression reduces to true. When the satisfiability of the constraint c cannot be determined, the expression blocks until such time as either the constraint or its negation is entailed by the store. In brief, implication involves asking c and then telling E if the ask returns true. Hentenryck Simonis and Dincbas [70] discuss implication and cardinality in the context of scheduling and VLSI testing.

3.3.3 Dynamic change and monotonicity

Whereas the cc() framework provides mechanisms for handling concurrency, it does not provide explicit techniques for handling time dependency. Some forms of temporal relationship are readily supported in cc(), through the use of blocking ask which can be used to implement sequences. However, certain applications, notably those related to scheduling and to dynamic modelling, require the use of more expressive constructs. This is not to say that such systems could not be implemented within the cc() framework: indeed the appropriate constructs could be built using concurrent constraint programming techniques, but they are not

provided explicitly in the cc() family of languages.

A standard technique used in constraint-based UIMS for modelling dynamic properties of the interface is to assert and retract constraints as the interface changes. The cc() framework assumes a store that is *monotonically refined* that is, once a constraint has been told, it remains satisfiable forever and cannot be retracted. Because of this, it would seem at first sight that the cc() framework is inappropriate for UIMS work. However, the use of *time-indexed variables* overcomes this apparent problem.

Duisberg [36] describes an animation system, Animus, which uses a global message queue to implement temporal dependency, thereby allowing the expression of dynamic behaviour in physical simulations. Interleaving and ordering can be expressed in Animus, using an event-based model of communication. This suggests that it could also be used for the specification of dialogue, though it has not been used for that purpose.

The system Kaleidoscope [47] uses time-indexed constraints to achieve a language that can be used in an imperative fashion. Constraints may be marked *always*, *now* or *during* (i.e. while some condition is true). Kaleidoscope provides a set of operators for expressing concurrency and sequence explicitly and uses a *committed-choice* method to implement constraint satisfaction. In this method, once a variable has been assigned a value in a given time-slot it cannot be changed again until the next interval. State in Kaleidoscope is represented by infinite streams, as in the language Lucid [6].

Features of procedural languages such as assignment and transfer of control are nonmathematical so do not lend themselves readily to formal proofs. Lucid treats variables as infinite sequences, indexed by a pseudo-time variable. Operators therefore apply to whole sequences. Most operators are applied pointwise to sequences, by pairing values with equal time indices. e.g

$X = Y + Z$

actually means

$X_0 = Y_0 + Z_0$

$X_1 = Y_1 + Z_1$

Certain operators do not apply pointwise. e.g 'asSoonAs'

$X \text{ asSoonAs } P = \langle x_j, x_j, x_j, \ldots \rangle$

where i is such that $p_i$is the first true value of P. The operators first and next are de-

fined as

first $X = <x_0, x_0, x_0, ...>$

next $X = <x_1, x_2, x_3, ...>$

where

$X = <x_0, x_1, x_2 ...>$

A similar form of time-indexing has also been used in the constraint language PROSE [13]. Attributes are kept in a time series, so that constraints may reference previous values. x(t) means the value of x at time t. Omitting the time reference is equivalent to using the current time value as t.

This use of time-indexed variables is similar in approach to the perpetual processes of Vulcan (Kahn et al [84]). It predates the cc() framework, but is interesting as a model for amalgamating concurrent logic with an object-oriented (actor-based) style of programming. While concurrent logic programming languages may be used to implement certain object-oriented idioms such as message passing and inheritance by delegation, it is awkward to use them to program in object-oriented style, because they lack a concise notation for these constructs. Vulcan attempts to remedy this deficiency. The main problem in using logic to model objects is that of providing persistent state.

Object systems based on concurrent logic (e.g. actor systems) use side-effect free processes to model objects in which persistence is achieved by the recurrence of the process with new parameters each time a state-change occurs. Persistence of reference is provided by the use of streams, shared queues of data which act as the medium of communication between processes. Thus, in concurrent Prolog, object identity is associated with a variable that contains a stream of messages. Senders pass a new stream to the process, which spawns other processes, replacing itself with a new process with the same functor and the new stream, which can then be used to access the new process. These are called *perpetual processes* because, although they terminate, they replace themselves with a copy of the original. The main problem with this technique is that it is necessary to copy the whole state of the object explicitly and to ensure that the stream is read-only (otherwise unification will result in the stream getting instantiated with arbitrary messages). Kahn et al describe the advantages and disadvantages of this approach.

> Since objects with state are not taken as a base concept, but are built out of finer-grained material, many variations on traditional notions of object-oriented programming are possible. These include object forking and merging, direct broadcasting, message-stream peeking, prioritized message sending and multiple message streams per object.
>
> In exploring these issues, we found that the implementation of objects was extremely verbose and fraught with sensitivity to small mistakes. We remedy this by creating a higher-level language with a syntax which succinctly captures the cliches used for object-oriented programming in concurrent logic programming languages.
>
> [84]

3.4 Solution techniques for qualitative constraints

In this section we consider a class of constraint problem in which the domains of the variables in the constraint net are not enumerated explicitly. Instead, the constraint net represents the allowed relations between variables. These relations are represented in the net by sets of labels on the arcs. Solution of this kind of problem relies on modified algorithms for path consistency. We will consider only the case of binary constraints. In this kind of problem the definition of consistency is based on a set of rules governing combinations of the basic constraints and a solution to the problem consists of a labelling of the constraint net such that each arc has a single label. As an example, consider the three relations <,=,> which may be used to define a constraint satisfaction problem over the real numbers. This set of relations has been described widely in the literature on constraint programming and forms the basis for one approach to temporal constraints (e.g. that of Vilain & Kautz [153]). The rules of combination governing these relations are as shown in Table 3.1.

First, here is a simple example of a problem defined using the constraints <,=,>. The associated constraint net and one of its solutions are shown in Figure 3.3. We use the brackets {} to show the set (disjunction) of relations which is allowed between the two variables. We will term the labels which represent the individual relations *basic labels* or *elements*; and the labels which represent disjunctions of allowed relations *compound labels*. Thus, in the following example,

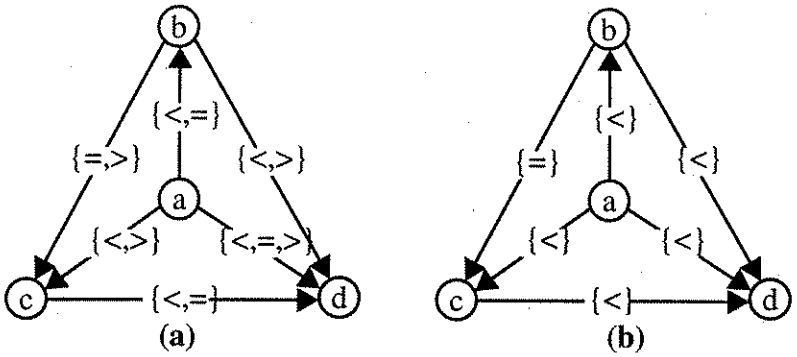


Figure 3.3 A constraint net using qualitative constraints (a), and a possible solution (b).

a rel b	b rel c	a rel c
<	<	<
<	=	<
<	>	<,=,>
=	<	<
=	=	=
=	>	>
>	<	<,=,>
>	=	>
>	>	>

Table 3.1 Composition rules for the relations <,=,>.

the compound label on the arc from a to b contains the basic labels '<' and '='.

{<,=} (a,b)
{<,>} (a,c)
{<,=,>} (a,d)
{=,>} (b,c)
{<,>}(b, d)
{<,=} (c,d)

Notice that a solution to the constraint net consists of a constraint net in which every arc has exactly one label. Also, the solution does not consist in finding actual values for the variables a,b,c,d, rather it consists in finding a *consistent scenario* which

defines the relation between the variables. That is to say the solution is qualitative, rather than quantitative.

A qualitative constraint net is said to be *maximally consistent* if every label in the net is part of a solution. In general, path consistency is not sufficient to find a maximally consistent net. For the system of constraints <,=,> there are consistent nets which are not maximally consistent. An example of such a net is shown in Figure 3.3. The net is path-consistent because every triple of arcs is consistent. However, there is no solution to the net in which a = d. Nonetheless, in this net, there are solutions in which a < d. One such solution is shown in the diagram.

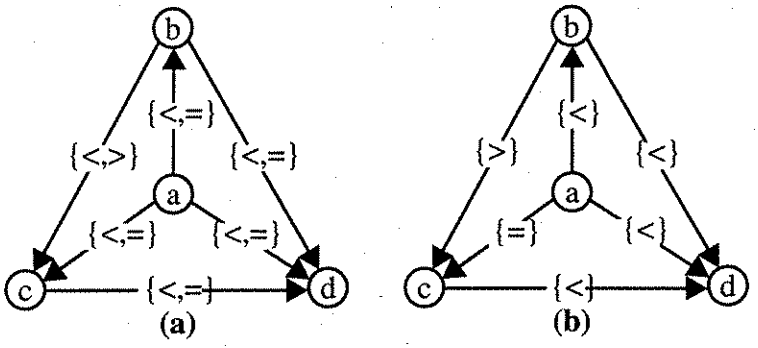


Figure 3.4 A path-consistent constraint net which is not maximally consistent (a). There is no solution in which a = d. The solution in which a < d is shown in (b).

a rel b	b rel c	a rel c
p	p	p
p	q	r
p	r	q
q	p	r
q	q	q
q	r	p
r	p	q
r	q	p
r	r	r

Table 3.2 Composition rules for a constraint system with relations p,q,r.

It is also possible for a system of constraints to permit consistent nets which admit of no solution. One such system of constraints is shown in Table 3.1. The rules permit any triple of relations in which each arc has a different label or all arcs have the same label. The constraint net shown in Figure 3.3 is path consistent but it has no solution. This can be seen by trying the two labels for the arc ac. If p(a,c), then the only solution for bc and ab is p(b,c) and p(a,b). However, p(a,b) forces q(b,d) and r(a,d). The constraints p(a,c) and r(a,d) together force q(c,d). Now the triple bcd is inconsistent, having the constraints p(b,c), q(c,d) and q(b,d). This just leaves the possibility for ac of r(a,c) and it can be shown by exhaustion of cases that here too an inconsistency results. Therefore, since there is no consistent label for ac, the network as a whole can have no solution.

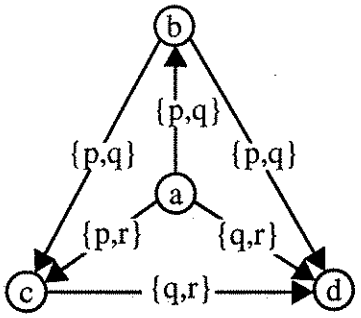


Figure 3.5 A path-consistent constraint net which has no solution. The composition rules for the relations p,q,r are shown in Table 3.1.

Thus, in general, path consistency is not enough to guarantee either maximal consistency or even the existence of a solution. As we saw in section 3.2.2 much research has focussed on solution methods which exploit the topology of the constraint graph. This research aims to define graph-theoretic properties which act as sufficient conditions to guarantee that a given network be soluble for *any* system of constraints. Typically, these properties are defined in terms of the width or connectedness of the graph and the solution is achieved by applying consistency techniques.

In our case, we can take a different approach since we are not interested in defining a general purpose constraint solver, but rather wish to find solution techniques for a specific system of constraints. We have not yet defined what that system is: that is the topic of the next chapter. However, many of the choices which we make with regard to this system will be influenced by consideration of the properties of a *system* of constraints which guarantee solubility. Thus, whereas

the work of researchers such as Dechter and Pearl, Freuder or Mackworth has been directed towards defining specific properties of constraint graphs which guarantee solubility for any constraint system, we are interested in defining specific properties of constraint systems which guarantee solubility for any constraint graph.

Our approach is to describe properties of a system of constraints which are sufficient to guarantee either the weak condition that path consistency implies the existence of a solution, or the stronger condition that any path-consistent net is maximal. The use of such systems is advantageous because there are polynomial time algorithms for path consistency, whereas the general problem of finding a solution to a constraint net takes time exponential in the number of nodes.

If path consistency is sufficient to guarantee the existence of some solution, then we can use the standard algorithms for path consistency whenever we are interested in knowing whether a solution exists. This result is relevant in the context of this thesis because we will use a system of spatial constraints embedded in a cc() -like language whose basic operations of **ask** and **tell** depend on the ability to check that a constraint net is soluble. If we design the constraint system so that path consistency guarantees solubility, then we may use standard techniques of enforcing path-consistency for the decision version of CSP (see section 3.1.2).

We have already seen that in the system <,=,> path consistency does not guarantee a maximally consistent net. However, van Beek shows that any path-consistent net constructed from these rules has at least one solution [11]. His proof is based on showing that the network may always be reduced to one containing only < and > to which topological sort provides a solution. However, van Beek's proof is ad hoc and does not generalize to other systems of constraints.

When we come to develop a system of spatial constraints in the next chapter, we will want to be able to show that there is a polynomial time procedure for the decision version of CSP. The remainder of this section is devoted to describing a set of sufficient conditions that in a given constraint system (a) every path consistent net has some solution and (b) every path consistent net is maximally consistent. The conditions which we develop are applicable to any system of binary constraints.

3.4.1 Required and forbidden labels

Given a qualitative constraint network, we may increase the information it

holds by telling additional constraints. The usual formulation for these constraints is to *forbid* a conjunction of labels, or, what is equivalent, to allow a disjunction of labels.

This results in the number of forbidden labels in the network increasing monotonically until eventually the network is entirely solved (every label has exactly one element which is not forbidden).

We described in Chapter 1 how our motivation for using constraints is so that we can represent the state of the user interface as a set of allowed display states. In this formulation, one interface state may be a subset of another and yet distinct from it. That is to say, we might have two interface states which were distinct in the sense that they corresponded to different domain model states, but every display state of the first was a display state of the second. This kind of situation was illustrated in Figure 1.1 of that chapter.

If we only allow constraints to forbid labellings then we will not be able to assert that an interface state holds, because it will always be possible to tell additional constraints that reduce the set of allowed display states. It would therefore be useful for us to be able to *require* labellings in the constraint graph. Once a label has been required, any subsequent tell which reduces the label will fail.

With required labellings, information is still added monotonically to the net, since the required and forbidden labels form disjoint sets which get increased by successive tells. However, the constraint net is ground when all labels are either forbidden or required. This means that the ground net may represent multiple states, which is exactly what we require.

Notice that this use of the term 'required' is somewhat different from the usage in the context of constraint hierarchies [16], where a required constraint is one that must hold in every solution and which can override other constraints. In our formulation, all we stipulate is that there should be some solution of the net for which the required label holds, not that it should hold in every solution.

In order to implement required labels, we need to be able to guarantee that every required label occurs in some solution of the network. This is, of course, the property of maximal consistency. If we can ensure that path-consistency is enough to guarantee maximal consistency, then we can implement required labels by the following scheme:

- When a label is required by a constraint tell, check that it is not forbidden and mark the label as required. If the label is already forbidden, the constraint tell fails.
- When a label is forbidden by a constraint tell, check that it is not required and that enforcing path-consistency does not result in the removal of a required label. If it does, then the constraint tell fails. If the constraint tell succeeds, then enforce path consistency.

We will give formal definitions of these operations in Chapter 5.

3.5 Conclusions

In Chapter 1 we described a model of the user interface which was based on the concept of a visual representation. In that chapter we also considered the architectural requirements for any interface which was to embody the model and we were able to say in general terms why constraint-programming was likely to be a useful tool in specifying interfaces based on such architectures. We have now come to the point where it is necessary to define the constraint system in detail.

The approach we will use is to embed a system of qualitative spatial constraints into a concurrent framework based on cc(). This is entirely novel in the context of UIMS work and there is therefore a strong obligation to justify the choice. In particular, we need to be clear about why the more standard approach, using local propagation, will not serve our ends.

The use of constraints provides a powerful means to specify declaratively properties of the user interface. The key areas where they may be useful are

- graphical layout;
- structural consistency;
- behavioural specification;
- control and communication.

The choice of algorithm used for constraint satisfaction strongly influences the utility of constraints in each of these key areas. Work in UIMS has used methods based on constraint propagation almost exclusively, mainly because these are easy to implement efficiently. The simplest of all methods to implement is the use of one-way constraints, as in Garnet or Rendezvous and these systems show that it is possible to implement a large variety of components using this idiom. However, the point of using constraints is not only to be able to produce a certain set of interface objects, but to simplify the task of programming these objects. The declarative nature

of constraints facilitates this task because specifications can serve directly as implementations, but the use of one-way constraints greatly reduces the clarity of constraint-based definitions and requires the designer to have detailed knowledge of the implementation of components before being able to predict and control their behaviour.

A second difficulty with methods of local propagation is that they do not adequately support constraints which express indeterminate values. In particular, those methods which have been tried in UIMS do not allow the designer to specify that the value of a variable lies within a range, without determining its value exactly: layout specifications are greatly simplified if this kind of constraint can be expressed, to say that two objects overlap, or that they intersect, for example. In this case, the constraint satisfaction algorithm restricts the range of constructs that can be expressed.

Another difficulty with the use of constraints in interface design is that the constraints usually govern a very general, low-level domain, such as the real numbers. In Garnet, for example, constraints have the full generality of Lisp code. Where there are no tools to build abstractions from these low-level constructs, the designer is obliged to provide excessively detailed specifications. Useful abstractions hide the details of implementation (by definition), but the use of one-way constraints limits the range of implementation detail that can be hidden. So successful abstraction requires the use of very general constraint solvers, which in practice are unable to provide the response times needed for interactive interfaces. The use of general constraint domains thus forces a compromise to be made between implementation efficiency and ease of expression. A guiding principle in the design of constraint systems should be that, where the full generality of the system is not used, it should not incur any implementation overheads. As Jaffar *et al* observe:

> We adopt the underlying philosophy that different kinds of constraints should be tackled by different algorithms. In particular, certain kinds of constraints may be special cases of certain others, and the cost associated with the more general algorithm should be paid only when these more general constraints are confronted.[82, p341]

This suggests the use of a range of constraint satisfaction techniques, with a narrowly defined constraint domain, specifically adapted for use in building interfaces. The cc() and CLP() systems provide semantics and operational frameworks for

defining constraint languages. Research with cc() to date has concentrated on providing constraint languages over finite domains and infinite trees and, within the CLP() framework, over the real numbers. One advantage of cc() in particular is that it is specifically adapted to the expression of concurrent systems and can be used to implement agent-based descriptions of systems. In chapter 2, the use of such descriptions as specification tools for UIMS was described and the possibility of using them to construct formal proofs of properties of the interface outlined. However, cc() is not in itself a language, it is a framework which must be instantiated by the addition of a domain-specific constraint solver, together with support for domain-specific objects and object-relations.

The choice of a system of constraints may be based upon the solubility of the resulting networks and the intended uses of the system. In particular, if we are concerned mainly with checking the solubility of the system, or with finding any possible solution, rather than enumerating all solutions, then a system of constraints in which path-consistency guarantees the existence of a solution is preferable.

Chapter 4 Spatial constraints

This chapter describes a set of constraints which may be used for the specification of graphical aspects of user interfaces. The description is based on a formalization of 2.5D space. There are several possible approaches to the formalization of space and these are considered in section 4.2. We show how one of these, the *mereological* approach, may be adapted to the specific needs of 2.5D interfaces in section 4.3. In so doing, we develop a series of increasingly powerful constraint systems for reasoning about topology, orientation (section 4.4) and shape (section 4.5).

4.1 Introduction

If *'a picture is (sometimes) worth ten thousand words'* [93] then the corollary is that descriptions of pictures are (sometimes) verbose. This gives rise to many difficult problems for the user interface designer, who has to give an account of the pictorial properties of the user interface and then relate them to a dynamically changing application domain. Most research in UIMS has concentrated its efforts on providing descriptions of the semantic links to the application and neglected the problem of describing the diagrammatic properties of the interface. The general approach to graphics is a reduction to the domain $\mathbb{R}^2$ (the 2D plane in real cartesian coordinates). While this approach is *complete* in the sense that it can describe any possible picture, it offers few abstractions to facilitate description.

Egenhofer makes the point that what is needed is a well-founded common vocabulary.

> While some terms may be specific to particular applications, in general all spatial relationships are founded upon fundamental principles of geometry. A consistent and least redundant approach requires that the concepts are identified at the geometry level in the form of a fundamental set of spatial relationships.
>
> [37]

Qualitative approaches are well suited to the definition of such fundamental relationships because they abstract away from the excessive detail which working in

absolute measurements requires. Each qualitative approach is characterized by a distinct calculus of graphical entities, relationships and properties. These can in turn be categorized according to the spatial transformations for which they retain their validity. For example, topological approaches describe properties which remain invariant under transformations which preserve adjacency relations (affine transformations). A review of these qualitative techniques is undertaken in section 4.2.

There is a recurrent theme in nearly all of the techniques presented below. They all start out with a theory of space from which a set of mutually exclusive binary relations between objects is obtained, a composition table is created for pairwise combination of the relations and then this is employed to compute the transitive closure of the network of relations. The algorithms for path consistency which were discussed in Chapter 3 provide a means to compute the transitive closure. We saw in that chapter that where the allowed relation between two objects is presented as a disjunction of basic relations, in general, path consistency does not guarantee a consistent solution to the network.

Such approaches when applied to arbitrary graphical objects start off with a formal account of the topology of the objects and then derive a set of distinct binary relations. The number of cases is generally large and very large composition tables are needed, since the size of the table is $O(n^2)$, where n is the number of distinct relations. This is undesirable for two reasons: the large number of binary relations are unintuitive and unwieldy for the user; and the size of the composition table makes computation of the graph closure expensive. One solution to this problem is to reduce the number of relations by grouping the primitive relations into disjoint sets that correspond to intuitive notions of space.

Two distinct ways to provide the theoretical description of space have been employed. These may be termed *point-set topology* and *mereology*.

Point-set topology treats objects as sets of points and has its foundations in geometry and topology, employing concepts such as simplices and open and closed regions [42]. Mereology has its foundations in logic. The idea here is to *axiomatize* the primitive entities and relations, without referring to any particular model of space. Typically, such theories are built on a single primitive relation, such as *part_of* or *connects* and one or more primitive sorts, e.g. *region*. Relations between objects are then derived in logic using the primitives.

The advantage of the mereological approach is that inference reduces to well-understood techniques of theorem-proving, which may be employed 'ready-made'. On the other hand, the fact that it is not directly grounded in any theory of space means that it has to be shown that the derived concepts have an intuitive plausibility and correspond to physical notions of space.

As it happens, there are extremely strong correspondences between work based on mereology and on point-set topology. Several authors have derived the same set of eight mutually exclusive topological relations, with only minor variations, from mereological and from point-set foundations. These eight relations have the required intuitive plausibility, since they correspond to easily recognizable, distinct arrangements of two regions. They are described below and shown diagrammatically in Figure 4.2.

Much work in this field has been motivated by the need to provide end-users with tools to manipulate spatial databases for purposes such as GIS (Geographical Information Systems). As such, the underlying mathematical foundation for any theory of spatial relationships is less important than the utility of the tools and concepts which it provides. The fact that authors working independently, from starting points as distinct as mereology and point-set topology, should arrive at virtually identical classifications of spatial relations supports this view: what they had in common was the same motivation.

This is not an argument for dispensing with the formal underpinning: as Mukerjee and Joe point out. [107]

> ... a formal representation, by providing a known domain over which it is complete, unburdens the designer of many of the problems involved in ensuring that his/her vocabulary is powerful enough to describe all the possible descriptions that can arise. This is particularly appropriate for acquiring new concepts, where the completeness of the vocabulary and capability of hierarchical abstraction become important.

There are two claims that are often embodied in such research, either implicitly or explicitly. One is that spatial relations are in some way fundamental to language and thought: a view which has been put forward strongly by Jackendoff [80]. The other is that the spatial relationships derived from point-set topology or mereology are in some way 'natural' in that they correspond to familiar or even innate concepts. However, no experimental evaluation of the validity of either of these claims exists. Anecdotally, it can certainly be argued that

spatial relations such as orthogonality and alignment are important means of conveying information through diagrams and other graphic representations. We will not attempt to refute nor to validate such claims: our general view on such psycho-perceptual issues is that the 'correct' choice of representation technique is still best left to the art of the designer. Our aim, therefore is to provide a framework which supports a variety of visual representations without making a commitment to a 'best' form of representation.

4.2 Qualitative spatial reasoning

Many of the layout problems encountered in interface design are based on an implicit rectangular frame, in which meaningful relationships between objects are derived from their relative positions, e.g. *above*, *below*, *left* and *right*. We will not attempt to account for this prevalence, since it is likely that it depends on a number of complex cultural, cognitive and historical factors. It is certainly *convenient* for the system implementor since it is easier to compute with rectangular bounding boxes than with arbitrary objects.

For example, a common convention is to represent ordering of objects in a list by a vertical ordering in which later elements in the list come below earlier ones. This is a simple and direct mapping between the domain semantics and the spatial layout. Aside from their widespread use in defining graphical layouts, rectangles and rectangular grids have the advantage that they are relatively simple to manipulate. Various methods of qualitative reasoning have been applied to rectangular layouts. Many of these derive from Allen's work on temporal relationships in which an algebra of one-dimensional intervals is defined [3]. Although the original motivation of this work was to model temporal relationships, the method is equally applicable to any interval in one-dimensional, continuous space. The method may also be extended to higher dimensions. The general solution of even the one-dimensional case is NP-complete, although by restricting the range of expressions used, it is possible to produce polynomial time solutions [11].

Before considering the two-dimensional case, we will review the simpler one-dimensional one. Allen considered the basic relationships between two one-dimensional intervals. These relations are based on the relative positions of the endpoints of the intervals. In relation to an interval, a point may be entirely before, coincident with the start, contained within, coincident with the end, or entirely after the interval. By considering the positions of the endpoints of one

interval with respect to the other, 13 distinct relationships may be defined. These are illustrated in Figure 4.1.

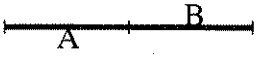
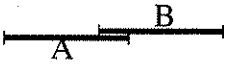
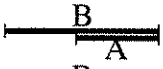
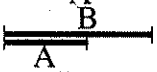
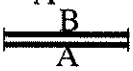
Relation	Name	Symbol	Inverse
	before	A {b} B	B {bi} A
	meets	A {m} B	B {mi} A
	overlaps	A {o} B	B {oi} A
	finishes	A {f} B	B {fi} A
	during	A {d} B	B {di} A
	starts	A {s} B	B {si} A
	equal	A {eq} B	

Figure 4.1 The 13 possible relations between two intervals, after Allen [3].

Partial information about the relation between any two intervals may be expressed as a disjunction of the basic relations, giving 2^{13} possible combinations. It is thus possible to represent a system of intervals by means of a network whose nodes represent the intervals and whose arcs are labelled with the set of permitted basic relations between the two intervals. A consistent scenario is a consistent labelling of the arcs, such that each arc is labelled with exactly one basic relation. This represents one of the allowed configurations of the system of intervals. An instantiation is achieved by assigning values to the intervals that satisfy a consistent scenario. Constraint satisfaction techniques, such as path-consistency may be applied to discover consistent scenarios (see section 3.2.3).

A somewhat simpler, albeit less expressive, approach is based on point algebra [153]. Using this approach, relations between intervals are expressed in terms of relations between the endpoints of the intervals. The relationship A {o} B (A overlaps B) may then be expressed as

$A^- < B^-$

$A^+ > B^-$

$A^+ < B^+$

where the start and end points of an interval A are written A^- and A^+.

It is possible to express all of the basic relations of interval algebra using point

algebra. However, point algebra cannot express the same range of partial information as interval algebra. In particular, it cannot describe situations in which an interval is constrained to lie within a discontinuous range. So, for example, in order to say that two intervals, A and B, were completely disjoint using interval algebra one would use the relation

A {b,bi} B

That is, either A is before B, or B is before A. The same situation cannot be expressed in point algebra, because it involves a disjunction of relations between different pairs of points i.e.

$(A^+ < B^-) \vee (B^+ < A^-)$

and the network representation of the point relations implies a conjunction of the relations represented by the arcs.

Van Beek describes an $O(n^2)$ solution for the interval relations expressible in point algebra [11] and argues that this restricted version of interval algebra is adequate for many temporal reasoning applications: we consider below the applicability of point algebra to the case of spatial reasoning.

Allen defines 13 possible relationships between two one-dimensional intervals. If regions are represented by their projections onto the horizontal and vertical axes, then there are $13^2 = 169$ distinct relationships between two regions. Additional relations are needed to handle points and lines and to take account of depth ordering and orientation. The size of the resulting set of possible distinct spatial relationships makes computation expensive and specification unnatural and unwieldy. It is therefore necessary to restrict the size of the set of relations by grouping together some of the primitive relationships.

4.2.1 Techniques based on point-set algebra

Egenhofer [37] considers binary topological relationships based on the intersection of boundaries and interiors of regions and develops a calculus based on simply connected regions (i.e. those whose boundary is a single, closed, non self-intersecting contour). He distinguishes three separate classes of spatial relation, as follows.

- **Topological:** these are invariant under translation, rotation and scaling.
- **Spatial order:** such relations rely upon the definition of a strict order between regions. In general such relations have a converse (e.g. above and below, left-of and right-of).
- **Metric relations:** exploit measurements of distance and direction.

In Egenhofer's approach, the relationship between two regions is specified by means of a table which shows the intersection of their boundaries and interiors, which may be either null or not null. The boundary of an object, A, is written ∂A and the interior A°.

There are 16 ($=4^2$) distinct ways to fill the table, but only 8 of these correspond to physically possible situations. These are illustrated in Figure 4.2. Two of these have inverse relations, so there are 6 distinct relations.

In an extension of this work [38], aimed at dealing with relations between objects of different dimensions (line–region, point–line, point–region), Egenhofer considers the intersection between interiors, boundaries and exteriors of objects (giving 9-cell tables). The 9-cell method results in the same eight relations between regions. Figure 4.2 shows their derivation from the 4-cell method. Egenhofer derives a composition table for these relations and illustrates the 64 cases which result.

Whereas Egenhofer only distinguishes empty and non-empty intersections, Clementini, di Felici & Oosterom [23] extend Egenhofer's approach to binary topological relations by considering the dimensionality of the intersection, giving 52 distinct relations. They then define five disjoint groupings of these relations, which they claim correspond to intuitive concepts: touch, in, cross, overlap, disjoint. These five groupings are used to define a calculus over regions, points and lines, which is complete in the sense that it can describe all of the basic cases.

Floriani, Marzano & Puppo also consider the dimensionality of the intersection in work which derives from consideration of the requirements of GIS (Geographical Information Systems). These have much in common with those of the user interface. Both are concerned with visual representation of information. They cite as fundamental:

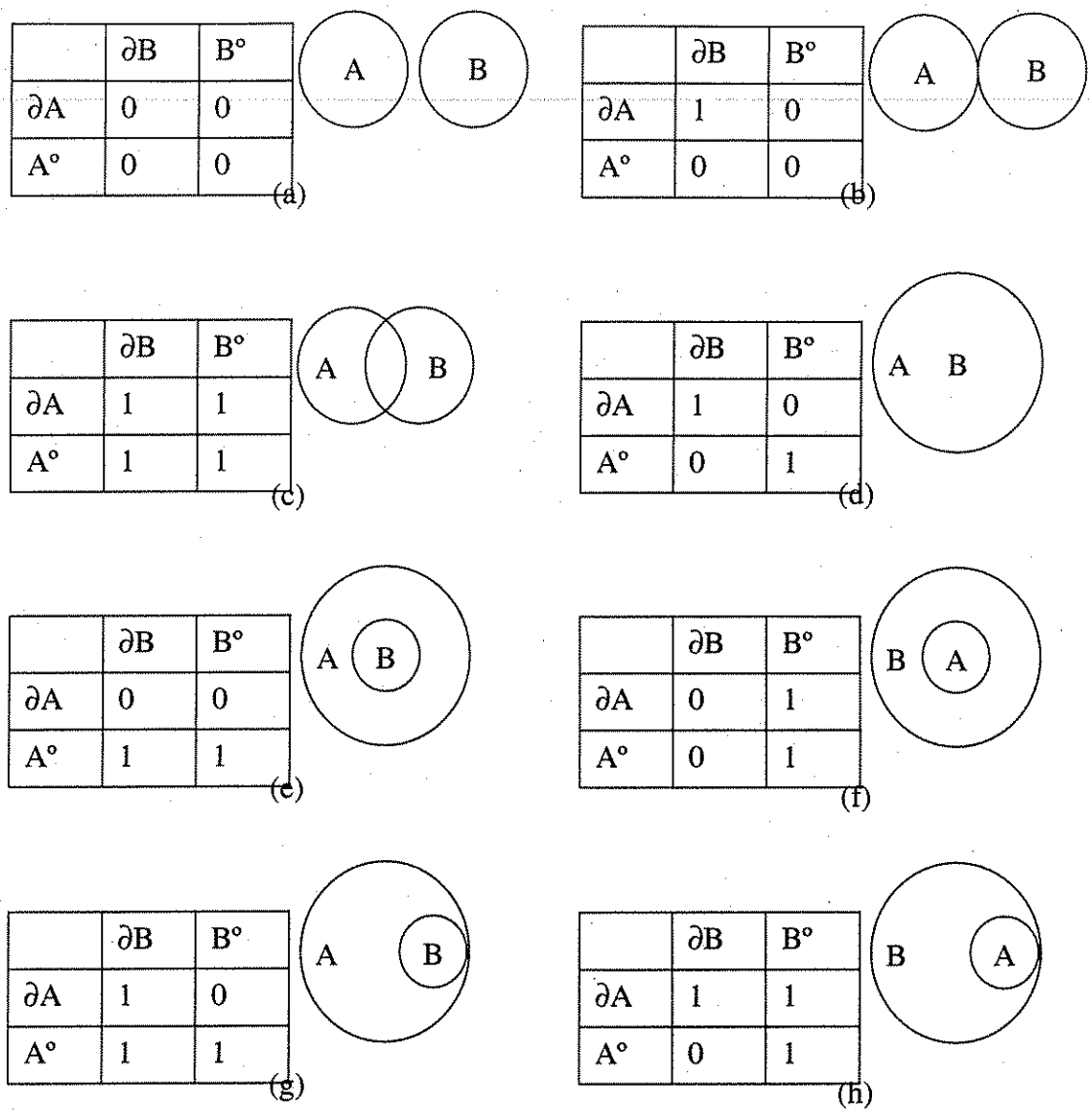


Figure 4.2 The eight feasible binary relations between regions, using Egenhofer's 4-cell intersection approach [37]. (a) disjoint(A,B); (b) meet(A,B); (c) overlap(A,B); (d) equal(A,B); (e) contains(A,B); (f) inside(A,B); (g) covers(A,B); (h) coveredBy(A,B). Relations (e) and (f) are inverse, as are (g) and (h).

- The definition and classification of spatial relationships.
- The development of efficient data structures and models for representing spatial layouts.
- The design of efficient algorithms.

The approach is object-based: the representation of spatial objects is independent of their position and uses a single data model, the Plane Euclidean Graph (PEG) to

represent points, lines, regions and topological relations. A spatial index is imposed on the PEG in order to improve the efficiency of queries (though at the cost of efficiency of update). The PEG defines objects by their boundaries and divides the scene into disjoint regions. Boundaries of objects have a dimensionality that is one less than that of the object (e.g. the boundary of a one-dimensional object, such as a line is a set of zero-dimensional objects, points). Various adjacency relations may then be defined in terms of the shared parts of the boundaries of objects. These can be categorized according to the dimensions of the shared parts.

In the PEG model, manipulation of the representation is transformed into the problem of manipulating the intermediate representation, the PEG. Many other qualitative approaches transform the spatial information into a graph. We consider later the possibility of using constraint networks as graph representations of spatial information.

An interesting variation on the intersection method has been proposed by Egenhofer & Sharma [39]. They consider its application to the plane $\mathbf{Z}^2$ (a grid of positive integers) so as to model properties of raster displays. In the case of raster displays, two forms of adjacency are possible: in one, object boundaries have points in common, in the other the boundaries have no points in common, but some pairs of points are adjacent in the raster grid.

Freksa suggests a means to group relationships, based on the idea of *conceptual neighbourhoods*. [49]. Two relationships are conceptual neighbours if gradual transformation of two objects in one relationship will result in the objects being in the other relationship, without any other relationship intervening. Thus, for example, meet and disjoint are conceptual neighbours, as are overlap and meet, but overlap and disjoint are not, since it is impossible for there to be a gradual transition from disjoint(A,B) to overlap(A,B) without meet(A,B) holding at some point in the transition. The 4- and 9-cell tables may be used to define the *topological distance* between two relations, in terms of the number of changes to the table that are made in passing from one relationship to another. The conceptual neighbours may be represented graphically, as in Figure 4.3.

4.2.2 Qualitative description of orientation

Hernandez [71] considers the interactions between topology and orientation. The treatment of topology uses the eight mutually exclusive binary relations described above (although he names some of the relationships differently). Orientation is described similarly in terms of a set of eight relations: front, back,

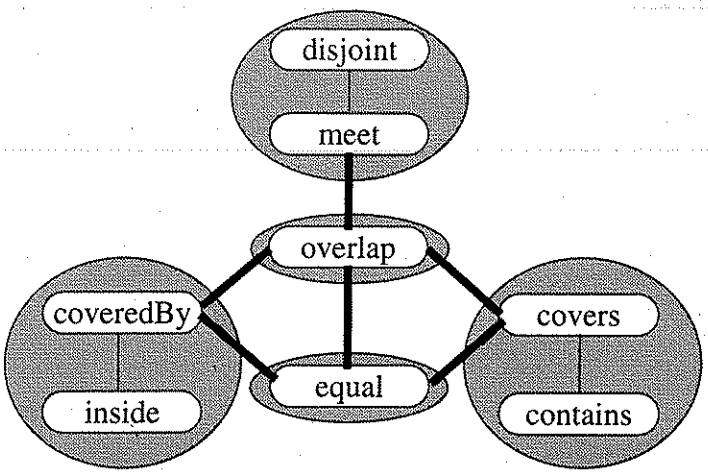


Figure 4.3 Conceptual neighbourhoods for the eight basic relations. Relations in the same subgroup (shown shaded) have a topological distance of 1. Intergroup relations (heavy lines) have a topological distance > 1 (after Egenhofer & Sharma [39]).

left, right, left-back, right-back, left-front, right-front.

These relations are specified with respect to a reference frame, which may be *intrinsic*, *extrinsic* or *deictic*. Intrinsic reference frames are specific to the particular reference object. Extrinsic frames are specified with reference to some external system and deictic systems relate to a particular point of view (see Figure 4.4). Thus, in Hernandez' system, relative position of two objects is given by a tuple:

<p, rel, r, f>

where p is the primary object, whose position is to be specified, rel is a pair of topological and orientation relations, r is the reference object and f is the frame to be used. The resulting set of relations is large.

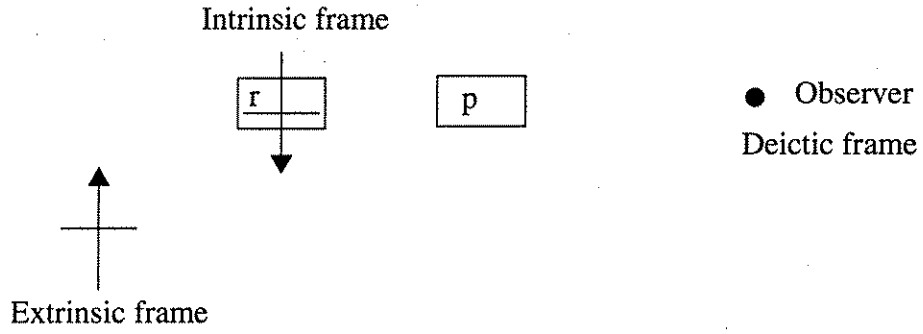


Figure 4.4 Intrinsic, extrinsic and deictic reference frames. In the extrinsic frame, p is to the right of r. In the intrinsic frame of r, p is to the left of r. In the deictic frame of the observer, p is in front of r.

Freksa considers orientation in terms of cardinal directions, based on *landmark*

points [50].

Mukerjee and Joe [107] propose a model of relative positioning which uses intrinsic frames of reference in combination with interval relations. Allen's method may be extended to n dimensions by considering the projections of objects onto each of the axes. This leads to 13^n distinct combinations of relations. Thus in two dimensions, objects may be projected onto the horizontal and vertical axes and the relations between the objects expressed as interval relations over each of the axes. This approach is complementary to those based purely on topology, since in general the topological relationship between the bounding boxes of an object will not correspond to that between the objects themselves. However, the method can represent information about relative position that is not expressible using topology alone. Moreover, Mukerjee & Joe claim that relations based on orthogonality and alignment have cognitive validity. However, rather than using the horizontal and vertical axes (an extrinsic frame) they argue that many relations are more naturally expressed in terms of intrinsic frames.

> A number of psychological test bear witness to the fact that the human cognitive process emphasizes accidental alignments ...
>
> ... orthogonality arises quite 'naturally' in human thought (e.g. left, right, front, back, east, north). One problem is that each object often has a different 'natural' orthogonal system, so that no one representation can model all of them. Another significant problem is that for non-aligned objects, the rectangular enclosures often overlap when the actual objects are disjoint.
>
> (Mukerjee & Joe [107], p722)

Their method depends on defining an intrinsic frame for every object, which specifies its orientation. A 'collision parallelogram' is then defined for each pair of objects and the relationship between the two objects is defined in terms of the interval relationship between the reference object and the primary object. In contrast with the methods discussed above, the inverse relationship r_{BA} cannot be determined from r_{AB}, therefore it is necessary to specify both in order to specify fully the relation between two objects. Further information is provided by specifying the quadrant of the frame of the reference object which contains the origin of the frame of the primary object. This is illustrated in Figure 4.5.

The full specification of position in this method is a 3-tuple
$<direction_{AB}, rel_{AB}, rel_{BA}>$

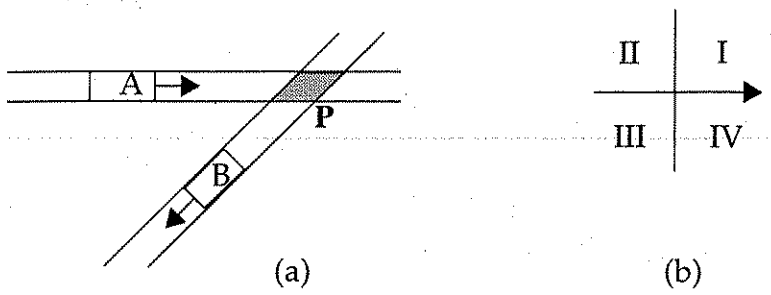


Figure 4.5 Mukerjee & Joe's method: (a) **P**, shown shaded, is the 'collision parallelogram' of **A** and **B**; (b) the quadrants of **A**'s intrinsic frame. Thus, $direction_{AB} = 4$; rel_{AB} = -- (**A** is entirely before **P** in **A**'s intrinsic frame); rel_{BA} = ++ (**B** is entirely after **P** in **B**'s intrinsic frame).

which results in 676 (=4x13x13) distinct binary relations, or 676^2 entries in the composition table. The size of the resulting table may be reduced by exploiting symmetries in its structure, nonetheless the number of distinct relations is unwieldy for the user and computationally expensive.

Mukerjee & Joe additionally propose the addition of operators which achieve some of the expressive power of using a full metrical system based on $\mathbf{R}^2$. The *reflection operator*, causes a reflection about a 45° line and is used to compare orthogonal distances. The *flush translation operator* translates one object to be flush with another, so that their dimensions may be compared. The *translation operator* causes the object to translate a given number of times along its own dimensions and so can be used to define aspect ratios.

4.2.3 Mereology

Egenhofer's method and its extensions are based on point-sets, in which regions, lines and points are formalized as sets of points, with intersection defined as set-intersection. An alternative foundation for spatial calculi originates in work done by BL Clarke [21, 22]. This is based on a *mereological* calculus: i.e. one that deals with part-whole relations. In contrast with classical mereology which uses relations *proper-part*, *overlap* and *disjoint* as primitives, Clarke's calculus is based entirely on a single relation, *connection*. Vieu [152] extends the calculus with operators that enable relative distance and orientation to be expressed. One advantage of this method is that points are treated as sets of individuals (and, hence, not as points in the classical sense) that may be refined to any level of granularity, as required. An almost identical mereological approach was developed independently by Cui, Cohn & Randell [31]. They define the eight

disjoint relations of Figure 4.2 in terms of their axioms, along with a set of compound relations and then derive the composition table by theorem proving. Like Vieu, they extend the calculus with functions, which enable convex hulls and relative positions to be determined.

The underlying basis of the approaches which we have so far considered is point-set geometry, or some other geometrical model (e.g the Plane Euclidean Graph). The advantages of the axiomatic approach to spatial relations is that it makes no commitment to any underlying formalism. In this approach, the underlying object is the region, rather than the point. While point-set semantics provide one possible model for the formalism, this is not a presupposition. Well-known methods of inference from first-order predicate calculus may be used to reason about the system, once it is described axiomatically.

The set of spatial relations which is described in the next section and which forms the basis for the work in this thesis uses a mereological approach.

4.3 Topological constraints

4.3.1 Introduction

The previous section considered the two main approaches to topology: point-set and mereological. In this section we develop a set of topological relationships appropriate for the user interface that has similarities to some of the mereological approaches described above, but additionally allows the expression of properties related to visibility. The two main differences are that we consider depth ordering and define a relation between an object and the display surface where it is drawn.

In this section, we consider the application of the methods described above to the specific problem of describing properties of user interfaces. These methods originated in attempts to formalize properties of physical systems: qualitative reasoning about dynamics and answering queries about information stored in geographical databases are two typical applications of this work. There has been a tendency to create user interfaces which simulate properties of physical systems; or, rather, it might be more accurate to say that there has been a tendency for user-interface designers to describe their designs in terms of physical systems. The properties of the interface which correspond to the physical system are collectively termed the 'metaphor'. A widespread, but probably inaccurate, view is that interfaces which employ metaphors are easier to use. We do not intend to pursue this issue here: our concern is merely to describe spatial properties of the user interface; the question of how these properties relate to the usability of the

system is a much larger and more complex issue (see for example the discussion in section 1.1.1). Techniques of reasoning about physical systems certainly have relevance to the description of the user interface, but there are differences between 'real' space and the abstraction of space which is used in graphical user interfaces. For this reason, the methods described above need some modification before they can be of use.

The general outline of this section is that we will develop a series of formal abstractions for the interactive space of visual representations in the user interface. We broadly follow the threefold division of Egenhofer (section 4.2.1) into topological, ordered and metric spaces. However, our approach is distinctly different from his, being based on mereology, rather than point-set theory.

We will use the specification language Z. Most of the specifications in this chapter could equally well have been based on other formalisms (e.g many-sorted logic, as used by Cui, Cohn & Randall [30,31] and Vieu [152]). However, we will ultimately need to be able to define properties of a system which stores and answers queries about systems of spatial relations and so it is convenient to be able to use a notation which serves as a vehicle for defining both logical properties of relations and properties of systems. The Z conventions used broadly follow Spivey [135,136].

In section 4.3.2 we develop a formalization of 2.5D space. This formalization is based on a single relation partCover. We do not make any assumptions about the existence of regions. In particular, we do not require that regions be defined constructively: there are no operators for the construction of regions in the axiomatization. Therefore, the formalization does not describe an algebra of regions (in contrast with the work of Clarke and Vieu presented above). However, it is sufficient to define a number of topological relations between regions, in the style of Egenhofer.

In section 4.4 we extend the formalization so as to be able to describe properties related to position and distance. The development throughout is qualitative, and based on logic, rather than on arithmetic. The rationale for this is twofold: techniques of consistency can be used to solve constraint networks constructed from such relations, thereby avoiding some of the problems of working with real numbers; and, as has been described above in the preamble to section 4.2, the qualitative approach is better suited for the description of spatial relations than purely arithmetic approaches.

We then consider, in section 4.5, the effect of introducing a tiling of the 2D plane. At this point, arithmetic becomes unavoidable and we consider some aspects of quantitative approaches. The theme throughout is to develop a set of constraint systems for handling different aspects of spatial relations.

4.3.2 The relation partCover

The underlying theory of space is based on an ordered set of non-intersecting two-dimensional layers: i.e. it is 2.5D. *Regions* are elements of a single layer. An *atomic region* is one which cannot be subdivided further. This is another difference from the approaches described above, in which it was assumed that regions were infinitely divisible. One possible model for this theory is the rectangular grid of pixels in a raster display. However, we do not make a specific commitment to this model and thereby retain the ability to describe other systems.

We make a distinction between elementary regions and compound ones. Following Kramer [89], we use the term *geom* to refer to spatial objects in general. The calculus we are developing will not include operators for the composition of regions: in particular, we will not require that intersection, difference and union of regions should exist. Thus we need to treat compositions of basic elements as separate kinds of entities from the regions themselves. We will define a part-whole relation, but we do not assume that every region has parts. The reason for taking this approach is that we wish to place as few restrictions as possible on the underlying set of basic symbols in any given application of the calculus. This point was made in section 1.1.3 when we referred to the set of symbols as the 'alphabet' of a visual representation. We will show how such operators might be defined, but they are not essential to the exposition of the rest of the thesis.

The formalization of the topological relations adopted here does not rely on distinguishing boundary and interior of objects. However, we will show how boundaries may be defined. We start with a basic relation between two regions:

partCover(a,b)

Intuitively, this relationship holds whenever region a is not entirely clear of region b. That is to say, either one region obscures the other, or the two regions appear to abut each other when viewed without regard to depth. Given that this relation can be calculated for all objects, then all of the other topological relations may be inferred. We first introduce the basic type

[Region]

which is the set of all regions in all possible spaces. Later we will introduce formal-

izations of particular kinds of space (topological, metric and so on), each of which will be characterized by a different set of possible regions.

Initially, we will make as few assumptions as possible about the nature of the space under consideration. Eventually, we will define quite restricted forms of space, such as those which can be mapped onto a grid structure, as in a pixmapped display. The basic relation partCover used to define topological 2.5D space is axiomatized as follows.

Topology

partCover: Region $\leftrightarrow$ Region

$\forall x$: Region $\bullet \neg$ partCover$^{+}(x,x)$

The property states that the relation partCover is irreflexive and forbids cyclic structures (see Figure 4.6). Cyclic structures are, of course, feasible in 3D space, but we are interested in modelling 2.5D space in which any set of objects has a total order of depth. We can justify this choice by observing that a great many window-based graphical user interfaces conform to this restriction.

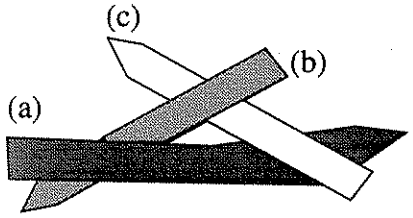


Figure 4.6 Mutually overlapping regions: partCovers(a,b), partCovers(b,c), partCovers(c,a). These are forbidden by the axiomatization given in 2.5D.

The system that we will develop permits a wide variety of interpretations in terms of physical models. For example, regions do not have to be simply-connected and we avoid premature commitment to any underlying physical representation.

We will first define some additional relations on the basis of this axiomatization and then consider how to restrict the definition of space to model different kinds of picture. We show how the notion of a layer may be developed, as an equivalence class of regions and consider relations between coplanar regions (i.e. those in the same layer) and connected regions.

The functions coverage and covering are introduced to simplify subsequent definitions. The coverage of a region is the set of regions which it partCovers. The covering of a region is the set of regions which cover it. We will continue with our definition of topological space, refining the schema as we go.

Topology

Topology

coverage: Region → **P** Region

covering: Region → **P** Region

$\forall$ x : Region •

coverage(x) = {y: Region | (partCover(x,y)} $\wedge$

covering(x) = {y: Region | (partCover(y,x)}

Two regions are equal when their coverage and covering are the same. We will write this as eq(x,y).

Topology

Topology

eq: Region ↔ Region

$\forall$ x,y : Region •

eq(x,y) $\Leftrightarrow$ (coverage(x) = coverage(y) $\wedge$ covering(x) = covering(y))

Depth-ordering relations may be defined in terms of the transitive closure of partCover. For example, coplanar, over and under are as follows.

Topology

Topology

coplanar, over, under: Region ↔ Region

---

over = partCover $^{+}$

under = over^{-1}

($\forall$ x,y : Region • coplanar(x,y) $\Leftrightarrow \neg$ (over(x,y) $\vee$ under(x,y)))

4.3.3 Topology

The neighbours of a region are all those regions which it partCovers, or which are partCovered by it and the region itself. The coverage and covering of a region are

therefore subsets of its neighbours.

Topology

Topology

neighbours: Region → **P** Region

∀ x : Region •

neighbours(x) = {y:Region | eq(x,y) ∨ partCover(x,y) ∨ partCover(y,x)}

Now we can define j(x,y) (joint), d(x,y) (disjoint), u(x,y) (upon), h(x,y) (hide), as follows.

Topology

Topology

j, d, u, h, c: Region ↔ Region

(∀ x,y : Region • d(x,y) ⇔ ¬ partCover(x,y) ∧ ¬ partCover(y,x))

(∀ x,y : Region | partCover(x,y)•

u(x,y) ⇔ neighbours(x) ⊂ neighbours(y) ∧

h(x,y) ⇔ neighbours(y) ⊂ neighbours(x) ∧

c(x,y) ⇔ neighbours(y) = neighbours(x) ∧

j(x,y) ⇔ ¬ (neighbours(x)⊆ neighbours(y) ∨ neighbours(y) ⊆ neighbours(x)))

The inverse relations j^{-1} etc, apply when the second argument partCovers the first. Thus, j^{-1} (x,y) means the same as j(y,x). These relations may be further refined to distinguish cases of boundary contact. To simplify the definition, we define the interior set (intSet) of a region as the set consisting of those regions which are upon or hidden by it and the exterior set (extSet) as the set of all disjoint regions. We will use the terms *interior region* (resp. *exterior region*) to refer to members of the intSet (resp. extSet). That is, iff x is an interior region of y, then x ∈

intSet(y).

Topology

Topology

intSet,extSet: Region → **P** Region;

∀ x : Region •

intSet(x) = {y:Region | h(x,y) ∨ u(y,x)}

extSet(x) = {y:Region | d(x, y) }

Now we can define eight subcases of joint, by considering the intersections of intSet and extSet. StrictJoint (o(x,y)) is a form of joint in which both x and y have interior regions in common and interior regions which are disjoint from the other. (Compare this with the definition of joint which requires only that the regions have common and disjoint neighbours.)

The weak versions of hide, upon and disjoint are defined similarly in terms of the intSets, rather than the neighbours of the regions. Thus, weakly disjoint implies that there is no region which is upon, or hidden by both the regions. That is to say, the regions overlap, but contact between the regions is minimal. Weakly upon (wu(x,y)) implies that, whereas there is some neighbour of x disjoint from y, no interior region of x is disjoint from y. In this case, the contact between the regions

is as much as possible without x being upon y.

Topology

Topology

sj, wd, wu, wh, wc, br, bb, rb: Region ↔ Region

$\forall$ x,y : Region • j(x,y) |

$$
\begin{aligned}
sj(x,y) \Leftrightarrow\ & intSet(x) \cup intSet(y) \neq \varnothing \\
& intSet(x) \cup extSet(y) \neq \varnothing \\
& extSet(x) \cup intSet(y) \neq \varnothing
\end{aligned}
$$

$$
\begin{aligned}
wd(x,y) \Leftrightarrow\ & intSet(x) \cup intSet(y) = \varnothing \\
& intSet(x) \cup extSet(y) \neq \varnothing \\
& extSet(x) \cup intSet(y) \neq \varnothing
\end{aligned}
$$

$$
\begin{aligned}
wu(x,y) \Leftrightarrow\ & intSet(x) \cup intSet(y) \neq \varnothing \\
& intSet(x) \cup extSet(y) = \varnothing \\
& extSet(x) \cup intSet(y) \neq \varnothing
\end{aligned}
$$

$$
\begin{aligned}
wh(x,y) \Leftrightarrow\ & intSet(x) \cup intSet(y) \neq \varnothing \\
& intSet(x) \cup extSet(y) \neq \varnothing \\
& extSet(x) \cup intSet(y) = \varnothing
\end{aligned}
$$

$$
\begin{aligned}
wc(x,y) \Leftrightarrow\ & intSet(x) \cup intSet(y) \neq \varnothing \\
& intSet(x) \cup extSet(y) = \varnothing \\
& extSet(x) \cup intSet(y) = \varnothing
\end{aligned}
$$

$$
\begin{aligned}
br(x,y) \Leftrightarrow\ & intSet(x) \cup intSet(y) = \varnothing \\
& intSet(x) \cup extSet(y) \neq \varnothing \\
& extSet(x) \cup intSet(y) = \varnothing
\end{aligned}
$$

$$
\begin{aligned}
rb(x,y) \Leftrightarrow\ & intSet(x) \cup intSet(y) = \varnothing \\
& intSet(x) \cup extSet(y) = \varnothing \\
& extSet(x) \cup intSet(y) \neq \varnothing
\end{aligned}
$$

$$
\begin{aligned}
bb(x,y) \Leftrightarrow\ & intSet(x) \cup intSet(y) = \varnothing \\
& intSet(x) \cup extSet(y) = \varnothing \\
& extSet(x) \cup intSet(y) = \varnothing
\end{aligned}
$$

Weakly coincide (wc) is a non-transitive form of coincide. Boundary-boundary contact (bb), boundary-region contact(br) and region-boundary contact(rb) are degenerate cases, whose physical models correspond to relations between regions of lower dimensionality (points and lines). Boundary-boundary contact (bb) implies that no part of either region lies entirely over or under the other. Its intuitive physical model is coincidence between regions with no area (i.e two lines). Boundary-region and region-boundary can be understood similarly in

terms of borderline contact.

The ability to make the distinction between the subcases of j(x,y) depends on the level of detail present in the description of the scene. This contrasts with those approaches which are based on geometry, where any region can be subdivided into a set of primitive objects, such as simplices or points. For example, in Egenhofer's approach, based on 4- and 9-cell intersections, it is necessary for all regions to have well-defined boundaries. In the approach taken here, it is possible to define a set of parts which will act as a boundary, should it be desired to consider relations involving boundary topology (e.g. weakDisjoint, weakUpon), however it is not necessary to do so. Moreover, it is possible to control the granularity with which such relations may be distinguished. As has already been observed in the previous section, this is generally cited as a key motivation for the mereological approaches to topology.

4.3.4 Completeness and exclusivity of the basic relations

We will use the term *basic relation* to refer to a relation which belongs to a set of relations that is mutually exclusive and exhaustive. That is to say, for any two regions, exactly one of the basic relations holds between them. We have already alluded to this idea in the development of the relations. The choice of which set of relations to employ is to some extent arbitrary, since it is possible to group individual relations (we do this in the next section when defining the *visually distinct* relations) and to subdivide them (as we have already done when defining the subcases of the relation joint). In this section we show that the set of basic relations which we have defined is mutually exclusive and exhaustive and explore some of their other properties.

We have defined a number of relations in terms of the single relation, partCover and the complement operator. These are shown in the tables below (Table 4.1, Table 4.2 and Table 4.2). The tables also indicate if each relation is reflexive, symmetric or transitive.

To keep the tables compact, the inverses of the non-symmetric relations are not shown, but they have identical entries to their counterparts.

Name	Relation	Ref	Sym	Tra
equal	eq(x,y)	*	*	*
disjoint	d(x,y)		*	

Table 4.1 Relations possible when x and y do not partCover each other.

Name	Relation	Ref	Sym	Tra
hide	h(x,y)			*
upon	u(x,y)			*
coincide	c(x,y)			*
joint	j(x,y)			

Table 4.2 Relations possible when x partCovers y.

Name	Relation	Ref	Sym	Tra
strictJoint	sj(x,y)			
weakDisjoint	wd(x,y)			
weakUpon	wu(x,y)			
weakHide	wh(x,y)			
weakCoincide	wc(x,y)			
boundary–boundary	bb(x,y)			
boundary–region	br(x,y)			
region–boundary contact	rb(x,y).			

Table 4.3 Subcases of joint(x,y).

We have therefore defined 24 different relations: there are two cases where one region does not part cover the other, (eq, d) and 11 cases when x partCovers y and the 11 inverses of these cases.

It is straightforward to show that the relations are mutually exclusive and exhaustive. For the most part this is clear from the definitions.

First of all we consider the four main cases when partCover(x,y), viz j(x,y), u(x,y), h(x,y), c(x,y). These are defined in terms of the neighbours of x and y. Given two sets, exactly one of the following propositions must hold

- one set contains the other as a proper subset;
- the sets are equal
- the sets are neither equal, nor does one contain the other.

These correspond to the definitions of the relations, so it is obvious that the four relations are exclusive and exhaust all possibilities for partCover(x,y). Similarly, it is clear that the subcases of j(x,y) are mutually exclusive and exhaustive. This follows

from the fact that all eight cases are defined by reference to the membership of the same three sets. Each of the sets may be either empty or non-empty, giving the eight possibilities which are ennumerated as the subcases.

Composition rules are inferences of the form

$rel_1(a,b) \wedge rel_2(b,c) \Rightarrow rel_3(a,c) \vee ... \vee rel_n(a,c)$

For example

$u(a,b) \wedge j(b,c) \Rightarrow d(a,c) \vee j(a,c) \vee u(a,c)$

We have seen in the Chapter 3 how such rules form the basis for the solution of constraint networks based on the spatial relations.

4.3.5 Visually distinct relations

The relations that we have so far defined are all distinct topologically, but some of them are indistinguishable visually. The relations coincide, hide and justHide are always indistinguishable visually (even given that other regions are present) and so we will refer to their disjunction as cover(x,y)

$cover(x,y) = co(x,y) \vee h(x,y) \vee jh(x,y).$

The relations cover, overlap, disjoint, justOverlap, upon and justUpon are mutually exclusive and visually distinct. Informally, they may be defined as follows:

- **disjoint**: neither region obscures the other and the projections of the regions do not touch;
- **justOverlap**: one region just obscures the other, but no region is upon both;
- **overlap**: one region partially obscures the other;
- **upon:** all parts of the nearer region lie upon the obscured region and all the edges of the obscured region are visible;
- **justUpon:** some region partCovers the first region, but not the second; every region upon the first object partCovers the second;
- **cover:** one region entirely obscures the other

The relation disjoint is reflexive. All the other relations depend on the depth ordering of the two regions and so have distinct inverses. They are illustrated in Figure 4.7.

It is worth pointing out some of the important points of the argument so far.

The topological relations which have been defined are based on a single primitive relation between regions partCover. The exposition is based on logic, rather than on geometry, which distinguishes it from approaches such as that taken by Egenhofer. So far we have made no commitment to any underlying

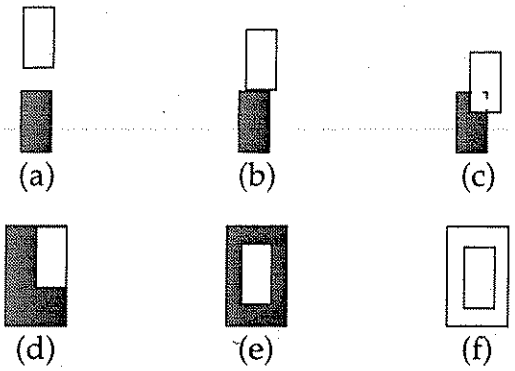


Figure 4.7 Visually distinct relations between regions w(hite) and g(rey). Region w is over region g. (a) disjoint(w,g); (b) justOverlap(w,g); (c) overlap(w,g); (d) justUpon(w,g); (e) upon(w,g); (f) cover(w,g).

representation of regions, thereby retaining flexibility in the implementation of the relations. The use of logic makes it possible to argue symbolically about the topology of the interface, without having to take account of the underlying geometric model. Given a set of regions, the relation between any pair may be computed, provided that the relation partCover is known for the whole set.

So far, we have not developed an algebra of space: we do not require the existence of regions formed by intersection and union of arbitrary regions, nor have we assumed that regions may be divided arbitrarily into parts. This contrasts with mereological approaches such as those of Clarke [21, 22] and Vieu [152], which develop Boolean or pseudo-Boolean algebras over space. (Vieu's approach leads to a pseudo-Boolean algebra because it lacks a null individual.) A full Boolean algebra would require an equivalence relation (which partCover is not), operators for intersection and union of regions (which we do not consider), null and universal regions (which we can define) and the complement operation (which has been defined).

It would not be too difficult to add the requisite components. We have a suitable candidate for the equivalence relation in coincident, which could be axiomatized to make it reflexive, symmetric and transitive. Definitions of intersection and union of regions could be constructed in a straightforward way in terms of the coverage and covering of the regions. We return to this point below, because it turns out that one possible model for such an algebra is a pixel-based grid.

As things stand, we do not consider the possibility of defining regions

constructively (as the sum and difference of other regions). The consequence of this is that the set of regions to be considered is limited to those which have been defined explicitly and may be as large or small as we wish (subject to the restriction that every region have its complement). The price that has to be paid for this simplicity is that, when there are few regions, only coarse topological distinctions can be made.

4.4 Relative position and orientation

In this section, we develop further the ideas presented in the previous section, in order to take account of horizontal and vertical ordering relations between objects. This will enable the expression of properties such as alignment and left-right ordering. The approach is based on defining a partial order between regions.

Alignment is widely used in graphic layout of the user interface, both as a means to improve legibility and to convey relatedness of information. The number of forms of alignment is large: some of them are illustrated in Figure 4.8. In the case of the rectangle, 'centre', 'top', 'left', 'right' and 'bottom' all have clear and unambiguous meanings. For the triangle, 'top', 'bottom', 'left' and 'right' have unambiguous interpretations, but 'centre' might refer either to the centre point of the bounding box, or to the centre of area, or even to the centre of the inscribed circle. Similarly, for the text,' bottom' might refer to the lowest point on the descenders, or to the base rule for the font. The developer of the interface may wish to provide default behaviours for alignment, but retain the ability to override the defaults in special circumstances, or to define new forms of alignment.

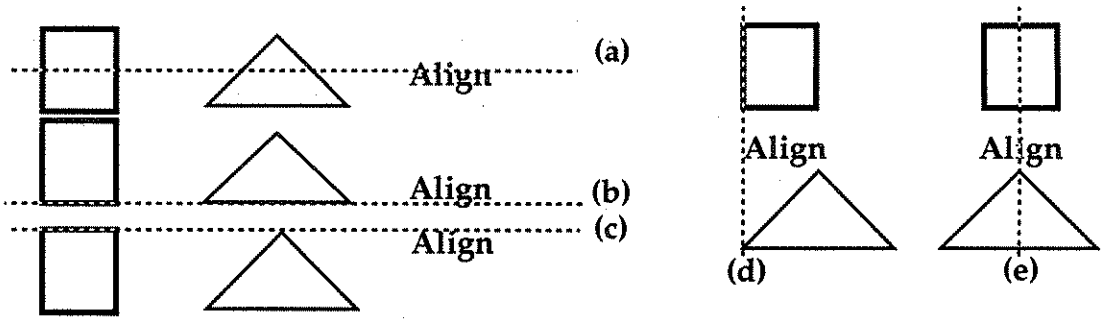


Figure 4.8 Some varieties of alignment: (a)Vertical centering; (b)Bottom; (c)Top; (d)Left; (e)Right

A similar problem of terminology arises in the context of relative positioning. Some objects may have a preferred orientation so that terms like 'left', 'right' and

so on are defined with respect to that orientation; however, the default is to define relative positioning with respect to some absolute orientation (typically the screen). An example of this is where a set of labels is to be applied to an inclined line. Text has an intrinsic orientation, so that terms denoting relative positioning are ambiguous: they may use intrinsic or extrinsic reference frames. We restrict descriptions of orientation to those which may be described using extrinsic frames.

Intrinsic and deictic frames are clearly useful for describing physical scenes: however, it is less clear what they add to the description of interfaces. Certainly, there is little use for deictic descriptions, since the viewpoint of the user is fixed. The exceptions would be interfaces which modelled physical structures, where it was necessary to give descriptions of a structure, as seen from different points of view. A better case can be made for using intrinsic frames, particularly when objects can be rotated. The use of construction frames for defining geoms makes this flexibility possible. Construction frames are templates which define *landmark* objects. The simplest construction frame defines an origin and bounding box for the object. We will see examples of this in Chapter 6.

We will develop a new schema, Orientation, to describe relative position. In order to define relative position, we require two new relations which can be applied to regions. These correspond intuitively to horizontal and vertical orderings. These are left(a,b) and below(a,b), for horizontally and vertically less than.

Orientation

left, right,
above, below: Region $\leftrightarrow$ Region

$right = left^{-1} \wedge above = below^{-1} \wedge$
$left = left^{+} \wedge below = below^{+}$

The ordering is not total, because the relations do not apply to all pairs of regions. Intuitively, left(x,y) implies that x is entirely to the left of y. We need horizontal and vertical alignment relations, but these cannot be defined entirely using the ordering relation, because left and below are only partial orderings and alignment implies an equivalence relation. There are various ways in which this difficulty can be overcome. One possibility is to introduce a set of atomic regions

which would effectively function as points, which would enable a grid to be constructed and alignment could be defined in terms of the grid. This is considered in section 4.5.

We will define 'weak' versions of left (*resp.* right, above and below), which are true when one object is at least partly to the left of (*resp.* right of, above, below) the second and the second is partly to the right of (*resp.* left of , below, above) the first.

Orientation

Orientation

weakLeft, weakRight, weakAbove, weakBelow: Region $\leftrightarrow$ Region

---

$\text{weakRight} = \text{weakLeft}^{-1} \wedge \text{weakBelow} = \text{weakAbove}^{-1} \wedge$

$\forall$ x,y,z: Region •

$\text{weakLeft}(x,y) \Leftrightarrow$
$(\text{left}(z,x) \Rightarrow \text{left}(z,y)) \wedge (\text{right}(z,y) \Rightarrow \text{right}(z,x)) \wedge \neg \text{left}(x,y) \wedge$

$\text{weakAbove}(x,y) \Leftrightarrow (\text{above}(z,x) \Rightarrow \text{above}(z,y)) \wedge (\text{above}(z,x) \Rightarrow$
$\text{above}(z,y)) \wedge \neg \text{above}(x,y)$

The weak orientation relations are weakly transitive: the composition of one with itself implies either the weak or the strong form of the relation. Thus

$\text{weakLeft}(a,b) \wedge \text{weakLeft}(b,c) \Rightarrow \text{left}(a,c) \vee \text{weakLeft}(a,c)$

Finally, we can define weak forms of alignment which apply to any pair of regions to which the orientation relations so far defined do not apply.

Orientation

Orientation

hAlign, vAlign: Region $\leftrightarrow$ Region

$\forall$ x,y : Region •

$\text{vAlign}(x,y) \Leftrightarrow \neg \text{left}(x,y) \wedge \neg \text{right}(x,y) \wedge$
$\neg \text{weakLeft}(x,y) \wedge \neg \text{weakRight}(x,y) \wedge$

$\text{hAlign}(x,y) \Leftrightarrow \neg \text{below}(x,y) \wedge \neg \text{above}(x,y)) \wedge$
$\neg \text{weakBelow}(x,y) \wedge \neg \text{weakAbove}(x,y)$

Notice that hAlign and vAlign are not transitive and only require there to be an overlap between the projections of the regions onto the corresponding axes. By considering parts of objects, we can achieve finer discrimination between configu-

rations of regions, as with the family of relations derived in the previous section.

We now consider the relationship between Topology and Orientation. The rules will enable us to construct a composition table: i.e. a set of rules of the form

$rel1(x,y) \wedge rel2(y,z) \Rightarrow rel3(x,z)$

which will be used to solve systems of constraints based on the relations.

SpatialRelations

Topology

Orientation

$(\forall\ x,y: \text{Region} \bullet$

$left(x,y) \vee above(x,y) \Rightarrow clear(x,y) \wedge$

$upon(x,y) \vee cover(x,y) \Rightarrow hAlign(x,y) \wedge vAlign(x,y))$

We summarize the orientation relations in Table 4.2 and Table 4.2.

Name	Relation	Ref	Sym	Tra
left	l(x,y)			*
right	r(x,y)			*
weakLeft	wl(x,y)			
weakRight	wr(x,y)			
vAlign	va(x,y)	*	*	

Table 4.4 Horizontal orientation relations.

Name	Relation	Ref	Sym	Tra
above	a(x,y)			*
below	b(x,y)			*
weakAbove	wa(x,y)			
weakBelow	wb(x,y)			
hAlign	ha(x,y)	*	*	

Table 4.5 Vertical orientation relations.

4.5 Shape, grids and constructive definition of regions

In this section we consider some ways in which the system of spatial relations

might be extended. In subsequent chapters, we will only use the relations so far defined. However, it is interesting to explore possible extensions of the approach.

We now consider how we might define an algebra based on the spatial relations so far defined. We will show that one possible model for the resulting system is a rectangular grid.

First, we present the requirements for a system to be a Boolean Algebra.

- A set of elements (in this case, the set Region).
- An equivalence relation (i.e. a two place relation which is reflexive, symmetric and transitive).
- An inversion operator (in this case, the complement operator).
- Two binary operators, which we shall designate '+' and '*', satisfying the axioms below, such that the elements x+y and x*y exist for all elements x,y. These are associative, distributive and commutative.
- A null element and a universal element, which we shall designate null and inf.

We will define a relation ≈ which will hold whenever two regions exactly coincide and which will serve as the equivalence relation.

The null element is required to be a member of the set of elements (Region) and the definitions of '+' and '*' must be such that

$x * \sim x \approx null$

$x + \sim x \approx inf$

The region, inf, partCovers no other region. The complement of inf is null. We will define the projection of a region as the parts of inf which lie in the extent of the region. It will be convenient to distinguish two subsets of Region: Viewable regions and GridPlane regions. Together, they include all elements of Region. We will define the resulting space as a Grid. It will need to use both topological and positional relations. So we will first of all consider the combination of Topology and Orientation. These together give us the essential requirements for 2.5D space, having both depth and position ordering.

$2.5D \triangleq [SpatialRelations \mid \forall x,y:Region \bullet (left(x,y) \Rightarrow d(x,y)) \wedge (above(x,y) \Rightarrow d(x,y))]$

We will now consider how to construct a grid in 2.5D space.

Grid

2.5D

inf, null : Region

Viewable, GridPlane : P Region

projection: Region → P Region

Viewable ∩ GridPlane = ∅

Viewable ∪ GridPlane = Region

~inf = null

GridPlane = parts(inf)

Viewable = {x:Region • partCover(x,inf)}

projection = extent ▷ GridPlane

projection(null) = ∅

4.5.1 Parts and proper parts

We now define two important relations: part and proper Part. The function parts will be useful in subsequent definitions. We will define

$X \subseteq_1 Y == X \subseteq Y \land X \neq \emptyset$

That is, X is a non-empty subset of Y.

Topology

Topology

part: Region ↔ Region;

properPart: Region ↔ Region;

parts: Region → **P** Region

∀ x,y: Region | •

part(x,y) ⇔ coverage(x) $\subseteq_1$ coverage(y) ∧ covering(x) $\subseteq_1$ covering(y)

properPart(x,y) ⇔ part(x,y) ∧ ¬ part(y,x)

parts(x) = {z:Region | part(z,x)}

We make the stipulation that x is non-empty, because we will need to deal with the null region, which is partCovered by no other region and partCovers no other region. We do not want null to be part of every region. The treatment of the null region is

generally problematic in axiomatic definitions of space (for example Vieu simply omits it from his calculus [152]).

A proper part is a part which is not equal to the whole. The transitivity and reflexivity of part follow trivially from the definition. i.e.

$\vdash$ part = part$^+$

which follows directly from the transitivity of $\subseteq_l$ in the definition of part. Also

$\vdash$ part(x,y) $\wedge$ part(y,x) $\Leftrightarrow$ eq(x,y)

can be derived trivially from the definitions of part and eq. We can also show that

part(x,y) $\vdash$ $\neg$ partCover(x,y) $\wedge$ $\neg$ partCover(y,x)

Since, from the definition of coverage,

part(x,y) $\vdash$ partCover(x,z) $\Rightarrow$ partCover(y,z)

however, from the definition of partCover

$\vdash \neg$ partCover(y,y)

Hence

part(x,y) $\vdash \neg$ partCover(y,y) $\wedge$ (partCover(x,z) $\Rightarrow$ partCover(y,z))

part(x,y) $\vdash \neg$ partCover(x,y)

and, by considering the definition of covering , we can prove similarly

part(x,y) $\vdash \neg$ partCover(y,x)

From projection(null) = $\varnothing$ and the fact that null is in the GridPlane, it follows that null has no parts. We will use projection to define an equivalence relation between regions, $\approx$, which implies that regions are coincident.

Grid

Grid

$\_ \approx \_$: Region $\leftrightarrow$ Region

$x \approx y \Leftrightarrow$ projection(x) = projection(y)

The relation states that iff two regions (fully) cover exactly the same parts in the GridPlane, then they are coincident.

We will use Σ for the sum of a set of regions. This will be defined as the minimum region in the GridPlane which contains the projections of all members of the set of regions. We place constraints on Region to ensure that Σ is defined for

every set of regions (though it may be null) and to ensure that it is unique.

Grid

Grid

$\Sigma : \mathbf{P}\,\text{Region} \rightarrow \text{Region}$

$\Sigma(\{\}) = \text{null}$

$\forall\ s: \mathbf{P}\,\text{Region}; r:\text{Region} \bullet$

$\text{part}(\Sigma(s),\text{GridPlane}) \wedge$

$\text{parts}(\Sigma(s)) \supseteq \text{projection}(s) \wedge$

$\text{parts}(r) \supseteq \text{projection}(s) \Rightarrow \text{parts}(r) \supseteq \text{parts}(\Sigma(s))$

The final predicate ensures that the sum of the regions is unique and minimum. We prove this informally as follows. Suppose that some part,r, of $\Sigma(s)$ also contains the projection of all the s. i.e.

$$\exists\ r:\text{Region} \bullet \text{part}(r,\Sigma(s)) \wedge (\text{parts}(r) \supseteq \text{projection}(s))$$

Then, from the final predicate,

$$\text{parts}(r) \supseteq \text{parts}(\Sigma(s))$$

and, since $\Sigma(s) \in \text{parts}(\Sigma(s))$

$$\text{parts}(r) \supseteq \{\Sigma(s)\}$$

$$\text{part}(\Sigma(s),r)$$

i.e.

$$(\text{part}(r,\Sigma(s)) \wedge (\text{parts}(\rho) \supseteq \text{projection}(s))) \Rightarrow \text{part}(\Sigma(s),r)$$

but, since

$$\text{part}(r,\Sigma(s)) \wedge \text{part}(\Sigma(s),r) \Rightarrow r = \Sigma(s)$$

we can conclude that

$$(\text{part}(r,\Sigma(s)) \wedge (\text{parts}(r) \supseteq \text{projection}(s))) \Rightarrow r = \Sigma(s)$$

The operators '+' and '*' can now be defined in terms of the union and

intersection of the projections of the regions.

Grid

Grid

$\_ + \_$,

$\_ * \_$: Region $\leftrightarrow$ Region

$x + y = \Sigma(\text{projection}(x) \cup \text{projection}(y))$

$x * y = \Sigma(\text{projection}(x) \cap \text{projection}(y))$

The relations + and * will enable us to define shapes constructively: that is to say, we will be able to define a region by specifying its parts.

Informally, a grid is a set of elementary regions which can be used to identify other regions uniquely. So we will define a GridElement as a part of the GridPlane which has no proper parts, and then require every combinationof grid elements to produce a unique region when summed.

Grid

Grid

GridElements: $\mathbf{P}$ Region

$(\forall\ g{:}\text{GridElements}; r{:}\text{Region} \bullet \text{part}(r,g) \Rightarrow r = g) \wedge$

$(\forall\ r,s{:}\ \mathbf{P}\,\text{GridElements} \mid \Sigma(r) = \Sigma(s) \Leftrightarrow r = s)$

In a rectangular grid, the alignment relations between GridElements are transitive. There is also a well-defined order between aligned elements. The simplest way to achieve this is by a mapping onto $\mathbb{N}^2$. So, finally, we arrive at the definition of metric spaces. The GridElements will correspond to points and be ordered by the mapping. Throughout the development we have avoided the use of quantitative descriptions and explicit reference to point-set topology: our intention in so doing has been to avoid developing a system which was committed to a particular form of implementation. However, we have now come to the end of the cases which may be dealt with without recourse to number. We need to invoke at least the integers (if not the reals) to describe absolute location, relative size and shape.

This final step involves the definition of a function coord to reference points (i.e. GridElements), given a location. We will axiomatize it so that it conforms to the

appropriate orderings. The point created will be in the grid plane, but we can also refer to points in other planes by using the equivalence operator '≈'. We will refer to the resulting system as Metric.

Metric

Grid

coord: $\mathbb{N} \times \mathbb{N} \rightarrow$ Region

$(\forall\ x1,y1,x2,y2: \mathbb{N} \bullet$

coord(x1,y1) ∈ GridElements

hAlign(coord(x1,y1),coord(x2,y2)) ⇔ y1 = y2

vAlign(coord(x1,y1),coord(x2,y2)) ⇔ x1 = x2

left(coord(x1,y1),coord(x2,y2)) ⇔ x1 < x2

above(coord(x1,y1),coord(x2,y2)) ⇔ y1 < y2

Now we can show how to define different shapes. We are going to need some primitive shapes to work with, so we shall choose line and point as the basic building blocks. Points are regions equivalent to GridElements. We define the endpoints of the line as those parts of the line which are points and which lie entirely above, below, left or right of the other parts. The relation endpoint (line, point) associates lines with their endpoints.

Metric

Metric

line _, point _ : $\mathbb{P}$ Region

endpoint : Region ↔ Region

---

point(p) = ran (GridElement ◁ ≈)

(∀ r, l :Region | line(l) • part(r,l) ⇒ line(r) ∨ point(r))

∀ l, p: Region | point(p) ∧ line(l) •

endpoint(l,p) ⇔ part(p,l) ∧
((∀ x: Region | p ≠ x • properPart(x,l) ⇒ left(p,x)) ∨
(∀ x: Region | p ≠ x • properPart(x,l) ⇒ right(p,x)) ∨
(∀ x: Region | p ≠ x • properPart(x,l) ⇒ above(p,x)) ∨
(∀ x: Region | p ≠ x • properPart(x,l) ⇒ below(p,x)))

Now we define properties related to structures formed out of lines and points. Two lines are linked if two of their endpoints coincide. The predicate path, is true of a sequence of lines if they are linked so that adjacent elements of the sequence are

linked and each element is linked only to its predecessor and successor in the sequence. If the first and last elements are also linked, then the path is a closedPath.Finally, we define a relation between ClosedPaths and regions called polyline, whose members are mappings from a ClosedPath to a region whose boundary is exactly that path.

Metric

Metric
linked : Region ↔ Region
path_, closedPath: **P** seq Region
polyline: seq Region ↔ Region
linked = { l1, l2 : Region | line(l1) ∧ line(l2) ∧ (∃ p1, p2: Region | p1 ≈ p2 • endpoint(l1,p1) ∧ endpoint(l2,p2))}
path = {s: seq Region | ran s ⊆ line ∧ (∀ i: **Z** | 2 < i <#s−1 • linked({s(i)}) = {s(i - 1), s(i+1)})}
closedPath = {p: seq Region | path(p) ∧ linked(first(p),last(p))}
polyline = {edges: seq Region; poly: Region | closedPath(edges) ∧ **Σ** {r:Region | boundary(r,poly) } = **Σ** {r:Region | part(r,edges) }

The aforegoing examples should be enough to show how we may define a whole variety of different shapes, using the basic framework which we have developed.

Whereas regions have unique depth, geoms may be associated with arbitrary shapes, spread across several layers, built up from a set of regions. The set of regions may overlap arbitrarily.

4.6 Conclusions

We have defined two qualitative systems for describing spatial relations. These relate to topology and to relative orientation. In the previous chapter, we considered how to solve constraint problems based on such systems. The basis for solution is a set of composition rules for the system of relations. We give a summary of the relations in Appendix A.

There are basically two different ways in which to give formal definitions of spatial relations. In one, we start off with an axiomatized set of primitive relations (or a single primitive relation) and derive further relations through logic. This is the approach which we have adopted in this chapter. The alternative approach is

to derive the system of spatial relations from a theory of geometry. We have seen that both approaches have led to essentially the same set of spatial relations for 2D space. In this chapter we gave a formal definition of a system of spatial relations that is suited to the description of 2.5D space.

The main aim of this chapter was to describe a basic set of 2.5D spatial relations, together with rules for their composition. In the previous section (section 4.5) we considered possible extensions to the system. We were able to show that a number of interesting properties may be derived from the single relation, partCover. For example, it is possible to define relations of lower dimensionality such as edge contact. Additionally we can define structures such as grids and polylines. A full exploration of these possibilities is beyond the scope of this thesis, but the next two chapters will make use of the two systems of topological and orientation relations that were developed in section 4.3 and section 4.4. In the next chapter (Chapter 5) we show how to incorporate the relations into an agent-based notation, while in Chapter 6 we give some examples of how the relations contribute to the description of user interfaces with the notation.

Chapter 5 A constraint-based notation

This chapter shows how the spatial relationships which were defined in Chapter 4 may be incorporated into a language for describing visual representations in the user interface. Concepts related to constraint nets are formalized in section 5.2.In section 5.3 we describe how geoms may be defined and grouped together into structures, which are characterized by constraints between their parts. In section 5.4 we give a description of the behaviour of the constraint store and the basic operations of ask and tell. The next section (section 5.5) shows how we may define agents which communicate through the store.Finally, we summarize the notation (section 5.6) and give an overview of its role in specifying interfaces (section 5.7).

5.1 Introduction

This chapter is primarily concerned with showing how the spatial relations described in the previous chapter may be incorporated into a computational model for describing UIMS. The conception of the model owes much to the cc() framework for describing concurrent constraint-programming languages (see section 3.3.2). Its distinguishing feature is that computation is modelled as sets of agents communicating through a constraint store. The constraint store is refined monotonically, which means that once a constraint has been added to the store, it cannot be revoked. We have already alluded to the problem of modelling persistent objects and dynamically changing state within such frameworks: the method we shall adopt is to model persistent state using perpetual processes (see section 3.3). In Chapter 1, we reviewed some approaches to the description of visual representations, particularly those based on the idea of composing primitive languages. In this chapter we develop a notation which we claim can be used in such approaches; in the next chapter we give examples of its use. The semantics of the notation are based on a computational model which is also developed in the chapter.

We have already described a set of spatial constraints and shown how they may be solved. A notation based purely on the spatial relations between individual regions would be very awkward to use for describing the complex

structural relationships that exist in interface representations. So we will additionally need some way to model picture structure. To achieve this we define a constraint system based on finite trees. Such systems are well researched and there are known decision procedures for checking entailment and consistency of constraints based on them (see e.g the review of constraint systems in [126]). However, we need to show how the two constraint systems (structural and spatial) interrelate.

In order to motivate and clarify the discussion in this chapter, we will start off with a brief overview of the computational system which we are proposing. We will keep this introduction fairly informal since the rest of the chapter gives formal definitions of the components and operations of the system. Throughout the chapter we will develop a notation, which is then summarized informally in section 5.6.

The heart of the system is a constraint store, which contains partial information about the state of the interface. The store has three related components: a geom store which contains information about structural relationships between interface components; a spatial constraint store, which uses the system of spatial relations developed in the previous chapter to describe layout of interface components; and an external constraint store which models the represented application and mediates communication between external processes and the interface.

The behaviour of the interface is defined by using a system of agents which communicate with each other through operations on the store. This communication has two basic primitives: ask and tell, which respectively check if a constraint is entailed by the information in the store, and augment the store with extra information by adding a constraint.

The basic components which are to be described in this chapter are illustrated schematically in Figure 5.1.

An important property of the store is that it is *monotonically refined*. This means that although information can be added to the store, it can never be removed. This has the disadvantage that it is somewhat awkward to model dynamic behaviour, for which we will need to develop specialized idioms, but it has the important advantage of simplifying the task of reasoning about the behaviour of the store.

Closely related to the idea of monotonic refinement is that of *stability* of the

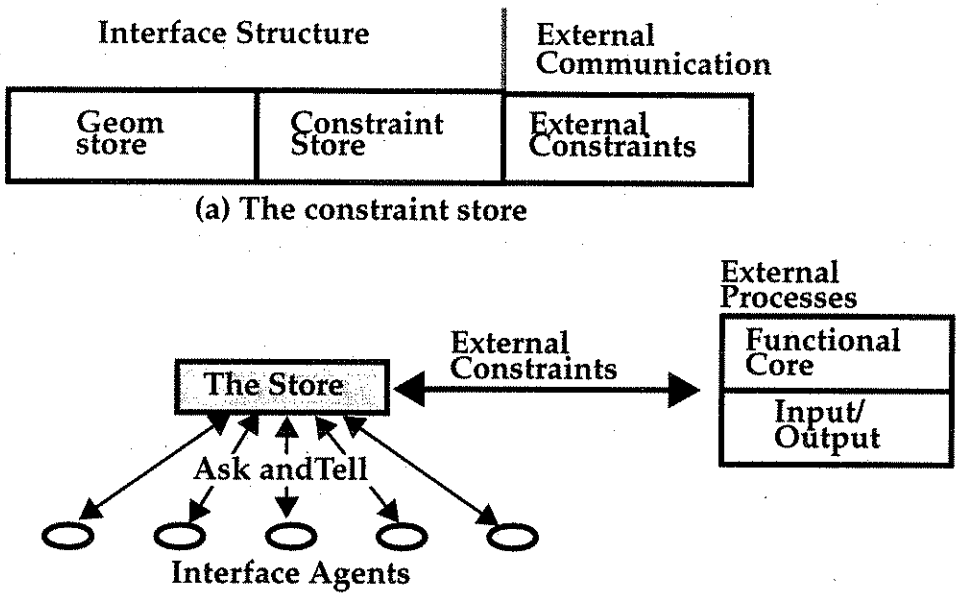


Figure 5.1 The underlying model for the notation developed in this chapter.

basic actions. Stability implies the following properties.

- If a tell (resp. ask) constraint has succeeded in a given store, it will always be possible to tell (resp. ask) the constraint, *in any subsequent store* .
- If a tell (resp. ask) constraint has failed in a given store, it will never be possible to tell (resp. ask) the constraint, *in any subsequent store* .
- An ask constraint succeeds whenever it can be told to the store without changing the store: i.e. success implies that the constraint is entailed by the store. It fails if the corresponding tell fails and it *suspends* iff telling the constraint would change the store.
- Only successful tells can change the store. Asks and failed tells never change the store.

First we define the properties of the spatial constraint store and its behaviour in response to the basic actions ask and tell.

In the standard approaches to finite domain constraint satisfaction problems, the nodes are labelled with sets of allowed values (unary constraints) and the arcs are labelled with allowed pairs of values for connected nodes (binary constraints). A solution to such a network is an assignment of a (single) value to each node, drawn from the allowed values, and which has an allowed pairing with each of the values assigned to its neighbouring nodes. We have already discussed such approaches in Chapter 3.

In order to apply this approach directly to spatial problems, it is necessary to define a finite set of possible regions to serve as the domain from which node values are assigned. There are several ways in which this might be done, one of which would be to define regions constructively in terms of a grid of basic atomic regions, as was outlined in the previous chapter. The resulting set of possible regions would then be finite, albeit large. While this approach is perfectly adequate for *formal* specifications of spatial relations, the large domain sizes which result make it unsuited to practical implementations of constraint solvers.

An alternative approach is to consider the allowed values for arc labels, as in the work by Allen on temporal relations. In this approach, nodes are not labelled explicitly with allowed values, but instead each arc is assigned a set of labels representing basic relations between the nodes. Associated with the network is a set of composition rules which specify allowed combinations of the basic arc labels. It is exactly such a set of composition rules that we formalized in Chapter 4. In this formulation, a *solution* is a labelling of the network such that every arc is labelled with exactly one of the basic labels and no set of arcs contradicts the composition rules. In this approach a solution to the constraint network is not an assignment of values to the nodes, but instead is an assignment of label values to the arcs so that we obtain for every pair of nodes a single label representing a relation which holds between those nodes. Having found a consistent labelling of the arcs (i.e. a having found a single consistent spatial relation for every pair of nodes), we then need to determine actual values for the nodes themselves (i.e. to assign a region to each of the nodes). The arc-labelling solution corresponds to a *qualitative* solution of the problem, in which the solution is given in terms of spatial relations. In general, the arc-labelling solution will underdetermine the node-labelling solution. That is, for any solution of the arc-labelling problem, there may be many possible valuations for the nodes.

We can therefore distinguish four aspects to the solution of dynamically changing spatial problems, such as occur in user-interface construction:

- The way in which objects and constraints are added to and removed from the problem. Here we are concerned with the dynamic aspects of the user interface and incremental techniques of solution are important.
- The set of all possible qualitative solutions to the network. Here we may be interested in asking if a certain relation is possible (i.e. the relation exists in some solution), or if a certain solution is implied (i.e. the relation exists in all possible solutions).
- The qualitative instantiation of the network, which corresponds to a single qualitative solution.
- The quantitative solution which corresponds to an actual state of the user-interface at some moment. This level is not described by the qualitative relations.

We will introduce the idea of a current solution, which represents the actual state of the interface. The interaction between the user and the application may be seen as an exploration of the solution space of the constraint network in which the user communicates with domain objects by imposing constraints on the interface objects.

5.2 Constraint nets

Our starting point for the definition of the constraint store is the set of spatial relations defined in the previous chapter and a set of region-valued variables.

5.2.1 Basics

We start out by defining a constraint net. We will assume as given sets

[NODE, LABEL]

First we define the composition rules for the relations. These are defined in terms of allowed triples of labels. Any set of rules will use a subset of the labels in which every label has a unique inverse, possibly itself.

CompositionRules

$$\begin{aligned} &\text{rules: } \mathbb{P}\,(\text{LABEL} \times \text{LABEL} \times \text{LABEL}) \\ &\text{inverse: LABEL} \rightarrowtail \text{LABEL} \\ &\text{labels: } \mathbb{P}\ \text{LABEL} \\ &\hline \\ &\forall\, l1, l2: \text{labels};\ \exists\ l3{:}\text{labels} \mid \bullet\ (l1,l2,l3) \in \text{rules} \\ &\forall\, l1, l2, l3: \text{labels} \mid (l1,l2,l3) \in \text{rules} \bullet (l2, l3, \text{inverse}(l1)) \in \text{rules} \end{aligned}$$

The first property states that there is at least one composition for every possible pair of labels. The second property ensures that reversing the direction of the arcs

does not affect the consistency of the network. Repeated application of the rule ensures that any triple will be consistent no matter which way the arcs are pointed.

We can now define a constraint net as a mapping from pairs of nodes (arcs) onto allowed sets of labels.

$Arc \cong [node1, node2: NODE \mid node1 \neq node2]$

$reverse: Arc \rightarrow Arc == (\lambda\ Arc \bullet (node2, node1))$

We will define three disjoint sets for every arc, the required labels, the unknown labels and the forbidden labels. Arcs whose nodes are not in the network map all labels to the unknown set.

ConstraintNet

$$
\begin{array}{l}
\text{CompositionRules} \\
nodes: \mathbb{P}\ NODE \\
forbidden, unknown, required, allowed: Arc \rightarrow \mathbb{P}\ LABEL \\
\hline
\forall\ Arc \bullet \\
\quad node1 \notin nodes \lor node2 \notin nodes \Rightarrow \\
\qquad forbidden\ \theta\ Arc = \varnothing \land \\
\qquad required\ \theta\ Arc = \varnothing \land \\
\qquad unknown\ \theta\ Arc = labels \\
\quad allowed\ \theta\ Arc = required\ \theta\ Arc \cup unknown\ \theta\ Arc \land \\
\quad \langle forbidden\ \theta\ Arc, required\ \theta\ Arc, unknown\ \theta\ Arc \rangle\ partition\ labels \land \\
\quad inverse\ (\!|forbidden\ \theta\ Arc|\!) = forbidden\ (reverse\ \theta\ Arc) \land \\
\quad inverse\ (\!|unknown\ \theta\ Arc|\!) = unknown\ (reverse\ \theta\ Arc) \land \\
\quad inverse\ (\!|required\ \theta\ Arc|\!) = required\ (reverse\ \theta\ Arc)
\end{array}
$$

Now we can define the notion of consistency. In Chapter 3 we considered algorithms for achieving various degrees of consistency. The essential feature of these algorithms is the removal of labels or values which are known to be impossible. In a triple of arcs, those labels which do not have *support* from other labels in the triple are impossible. A label has support if there are labels on the other two arcs which taken together form one of the allowed triples in the rule set. Another way of describing the same property is to say that a triple is consistent if it can be obtained

by merging together some subset of the allowed triples in the rules.

Consistency

CompositionRules

consistent : (**P** LABEL) × (**P** LABEL) × (**P** LABEL)

∀ s1,s2,s3 : **P** labels •

consistent(s1,s2,s3) ⇔

((∀ a:labels; ∃ b,c :labels | a∈ s1 ∧ b∈ s2 ∧ c∈ s3 • (a,b,c) ∈ rules) ∧

(∀ b:labels; ∃ a,c :labels | a∈ s1 ∧ b∈ s2 ∧ c∈ s3 • (a,b,c) ∈ rules) ∧

(∀ c:labels; ∃ a,b :labels | a∈ s1 ∧ b∈ s2 ∧ c∈ s3 • (a,b,c) ∈ rules))

Now we can define a constraint net to be *path -consistent* if every triple of allowed labels is consistent.

PathConsistent ≙ [Consistency; ConstraintNet|
∀ n1,n2,n3:NODE •
consistent(allowed(n1,n2), allowed(n2,n3),allowed(n3,n1))]

We will say that a subnet of a consistent net is *ground* if none of the arcs in the subnet has any unknown labels. An *instantiation* is a subnet which is ground and which has exactly one required label.

groundNet_ == {nodes: **P** NODE; net:ConsistentNet |
(∀ a:Arc | a.node1 ∈ nodes ∧ a.node2 ∈ nodes •
#net.unknown a = 0) }

instantiation_ == {nodes: **P** NODE; net:ConsistentNet |
(∀ a:Arc | a.node1 ∈ nodes ∧ a.node2 ∈ nodes •
#net.unknown a = 0 ∧ #net.required a = 1) }

Given a one-to-one mapping between nodes, i.e.

NodeMap == NODE ⤖ NODE

and a pair of constraint nets, we will say that the subnet defined by the domain of the mapping and the first net *instantiates* that defined by the range of the mapping and the second net if the first is an instantiation and all required labels of the first subnet are required in the corresponding arcs of the second. Once a subnet is known to instantiate another subnet, then it will continue to do so, whatever operations are applied subsequently. We can guarantee this because we know that the first subnet cannot be changed since it has only one allowed label on each arc.

instantiates == {map: NodeMap; i: ConsistentNet; c:ConstraintNet |
instantiation i ∧
(∀ a,b: Arc | map a.node1 = b.node1 ∧
map a.node2 = b.node2 • i.required(a) ⊆ c.required(b)) }

Note that only nodes in the domain and range of the instantiation are considered. Thus, if the node mapping is partial, we can check if one part of a net instantiates some other part. A subnet*weak ly* instantiates another if the required labels in the first are all *allowed* in the second. Note that this relation is not stable, since the set of allowed labels may get reduced by subsequent tells, thereby invalidating the instantiation. Therefore we will not use the relation directly as a constraint in the notation we are developing. However, it will be useful later in the definition of the consistent version of an inconsistent net.

weaklyInstantiates == {map: NodeMap; i: ConsistentNet; c:ConstraintNet |
instantiation i ∧
(∀ a,b: Arc | map a.node1 = b.node1 ∧
map a.node2 = b.node2 • i.required(a) ⊆ c.allowed(b)) }

The concept of an instantiation is closely related to that of a *solution* to a constraint net. Indeed, in many formulations of the constraint satisfaction problem, the definition of a solution is equivalent to that given above. However, we need to be able to distinguish two kinds of solution:

- the set of allowed display states which characterizes an interface state;
- an individual display state.

The reason for this is that we will characterize relations between interface objects in terms of a set of allowed primitive relations, rather than the particular relation that holds in any instantaneous display state. The mutual exclusivity of the basic spatial relations means that any set of regions in a display has a unique instantiation. That is to say, exactly one of the basic relations will apply between any pair of regions. Thus, instantaneous states of the display are instantiations. We considered the requirement for distinguishing between interface states and individual display states in the example presented in the Introduction (see Figure 1.1).

Another example shows a use for required constraints in modelling sets of display states. In a file manager, constraining a file so that it must be upon a directory causes that file to be copied to the directory. Required constraints may be used to show the write permissions relating to the copy. In order to show that the destination directory does not have write permission, we tell the constraint

Required {disjoint} (d1,f)

This does not prevent the file from being upon the directory in some instantiations, but it does prevent the file from being constrained to lie upon the directory: the required constraint means that there must be *some* instantiations in which the file is disjoint from the directory, even though there may be others in which it is not.

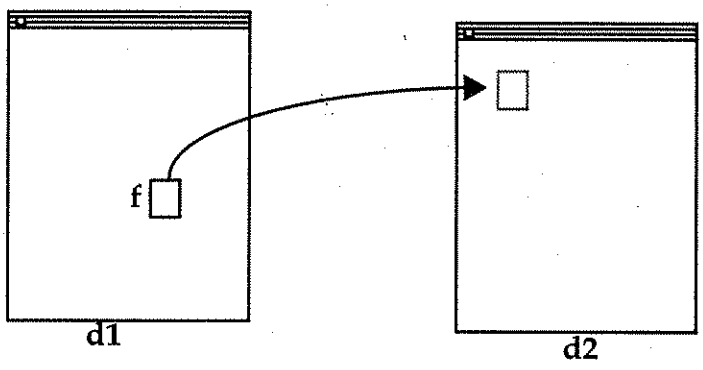


Figure 5.2 Use of required constraints. The file icon f is to be moved from directory d1 to d2.

Similarly we can show that the file may not be copied by requiring the constraint

Required {upon} (d1,f)

Given a (possibly inconsistent) constraint net, we often want to derive a consistent net from it. The consistent version of a net has the same set of (weak) instantiations as the former, but is also consistent. Additionally, every required label must be required in the consistent version.

consistentVersion: ConstraintNet → ConsistentNet

∀ map: NodeMap; i: ConsistentNet; c:constraintNet•

weaklyInstantiates(map, i, c) ⇔
weaklyInstantiates(map,i, consistentVersion(c)) ∧
c.required = consistentVersion(c). required

The consistent version of a constraint net is obtained by forbidding those unknown labels for which no instantiation exists. When an inconsistent net is made consistent, there is an increase in the information contained in the net.

A constraint net is *insoluble* if it has no instantiations.

InsolubleNet ≙ [ConstraintNet | θ ConsistentNet ∉ dom consistentVersion]

5.2.2 Operations on constraint nets

In this section, we look at some operations which can be performed on the constraint store. We will need these definitions when we describe the basic operations of ask and tell.

The merge of two constraint nets is the constraint net whose required (resp. forbidden) labels are those required (resp. forbidden) in either of the originals. Merge is only defined for pairs of constraint nets in which there are no forbidden labels in one which are required in the other.

$$\text{merge} : \text{ConstraintNet} \times \text{ConstraintNet} \rightarrow \text{ConstraintNet}$$

$$\forall\, c1, c2 : \text{ConstraintNet};\ a : \text{Arc} \bullet$$
$$(\text{merge}\ (c1, c2)).\text{required}(a) = c1.\text{required}(a) \cup c2.\text{required}(a)$$
$$(\text{merge}\ (c1, c2)).\text{forbidden}(a) = c1.\text{forbidden}(a) \cup c2.\text{forbidden}(a)$$

The merged net therefore contains all the information in both nets. If this information is contradictory, then the merge is undefined. The consistent merge is the consistent version of merge. We will denote it by '_+_'.

$$\_ + \_ : \text{ConstraintNet} \times \text{ConstraintNet} \rightarrow \text{ConsistentNet} == \text{merge} \ ; \ \text{consistentVersion}$$

We will define an infix relation $\_<_{CN}\_$ between constraint nets which is true when the first argument is entailed by the second. That is to say, when the second contains all the information in the first. We can check if one constraint net entails another by merging them: if the second net is equal to the merge, then it must entail the first.

$$\_<_{CN}\_ : \text{ConstraintNet} \times \text{ConstraintNet} == \{c1, c2 : \text{ConstraintNet} \mid \text{merge}\ (c1, c2) = c2\}$$

We will sometimes need to add new nodes to the net. The operation AugmentNet takes a ConsistentNet and an integer and adds that number of new nodes to the net. The arcs attached to the new nodes are all unknown. The operation therefore creates new, unconstrained nodes in the network. We will use

this operation to create local variables when we come to consider goal reduction.

AugmentNet

$$\Delta\text{ConstraintNet}$$
$$n? : \mathbb{N}$$
$$newNodes! : \mathbb{P}\ \text{Node}$$

$$newNodes! \cap nodes = \varnothing$$
$$\#newNodes! = n?$$
$$unknown' = unknown \oplus (\lambda\ Arc \mid node1 \in newNodes \vee node2 \in newNodes \bullet$$

5.3 Geoms and picture structure

We use the term geom to refer to visible objects. The difference between geoms and regions is that between objects and values. A region is a value which corresponds to a definite area in 2.5D space. A geom is a structure which identifies a set of possibly indeterminate parts. The parts may be, recursively, geoms or simply nodes in a constraint network, associated with some region value. The identity of a geom is partly derived from the regions it occupies and partly from its structure. In this section we will consider constraints over geoms and introduce some operators to create new geoms constructively. In terms of visual representations, geoms are the basic elements from which the representation is constructed.

Just as with the spatial constraint system, we will require that information about geoms is monotonically increased. Thus, once one geom is known to be part of another, it remains so forever.

Stuctures are sets of geoms, related by a set of constraints on their configuration. We introduce the set

[GEOM]

being the set of geom identifiers, and define some constraints on the structure of

geoms.

GeomStructure

$$
\begin{array}{l}
null\_, undefined\_, unused\_ : \mathbb{P}\, GEOM \\
parts: GEOM \leftrightarrow GEOM \\
labelledParts: GEOM \twoheadrightarrow (LABEL \twoheadrightarrow GEOM) \\
leaves: GEOM \twoheadrightarrow NODE \\
\hline
\langle null, undefined, dom\ parts, dom\ leaves \rangle\ partition\ GEOM \ \wedge \\
disjoint \langle unused, ran\ parts \rangle \ \wedge \ unused \subseteq undefined \ \wedge \\
\forall\ g:GEOM \mid (g \in dom\ parts) \bullet ran\ labelledParts\ g = ran\ parts \lhd \{g\} \\
\forall\ g: GEOM \bullet \neg\ parts^{+}\ (g,g)
\end{array}
$$

Geoms may be null, undefined, composites (having parts) or leaves (referring to a region-valued node in a spatial constraint network). Additionally there is a stock of 'unused' geoms, which can be used for operations which create new geoms. A geom structure is a set of rooted, directed, acyclic graphs. Null geoms do not have any parts. Undefined nodes may be parts of another geom, but do not themselves have any parts. The significance of the undefined nodes is that they are the only elements of the geom structure which may change. We will say more about this in a moment when we come to consider operations on the geom structure. In order to access the structure of the geoms, we introduce a labelling function onto the parts hierarchy. The schema states that all parts of a geom are labelled.

The following operation creates a geom with a given set of parts (all undefined).

$$
\begin{array}{l}
createGeom \mathrel{\widehat{=}} [\Delta GeomStructure;\ lab:\mathbb{P}\ LABEL;\ g!:GEOM \mid \\
\qquad dom\ labelledParts'\ g! = lab\ ]
\end{array}
$$

In the notation, we will write this as a list of labels in angle brackets. For example,

$$g = \langle top, bottom, left, right \rangle$$

says that g is a geom with four parts labelled 'top', 'bottom', 'left' and 'right'. The parts may be referenced with the usual dot notation for record structures, e.g.,

$$g.bottom$$

is the part of g which is labelled 'bottom'.

We will need to define rules for equality of geoms, so as to be able to apply

and check equality constraints on geoms. In doing this, we will consider as undefined any geom which has no parts, nor is a node value nor is null. Undefined geoms will only be considered equal if they are identical, or if they have been explicitly declared as equal. Otherwise, their relationship is undetermined. The rationale for doing this is that when we come to consider operations on geoms, we will only allow change to undefined geoms. Thus, once a geom has been assigned a set of parts, they are fixed forever. We do this so as to preserve monotonicity in the constraint system for geoms. Thus there are three possible mutually exclusive relations between any pair of geoms: equality, inequality or undetermined. Geoms with parts are equal if they satisfy the following conditions

- they have identical label sets;
- corresponding parts (i.e. parts with the same label) are equal.

Conversely geoms with parts are unequal if they have different label sets, or some pair of corresponding parts is unequal. Geoms which refer to nodes in a constraint net are equal if the nodes are equal.

First of all we give the axioms for equality and inequality.

GeomStructure

GeomStructure

equal, unequal, undetermined: GEOM $\leftrightarrow$ GEOM

$\langle$equal,unequal,undetermined$\rangle$ partition GEOM $\times$ GEOM $\wedge$

(equal = equal*) $\wedge$ (equal = equal $^{-1}$) $\wedge$

(unequal = unequal $^{-1}$) $\wedge$ (unequal ; equal = unequal)

Equality is transitive, reflexive and symmetric; inequality is symmetric. Now we extend the axiomatization to consider equality between geoms with parts.

GeomStructure

GeomStructure

$\forall$ g1,g2 : dom parts | # labelledParts g1 = # labelledParts g2 •

unequal(g1,g2)$\Leftrightarrow$ $\exists$ l: LABEL• unequal(labelledParts g1 l , labelledParts g2 l) $\wedge$

equal(g1,g2)$\Leftrightarrow$ $\forall$ l: LABEL• equal(labelledParts g1 l , labelledParts g2 l) $\wedge$

$\forall$ g1,g2 : dom parts | # labelledParts g1 $\neq$ # labelledParts g2 • unequal(g1,g2)

Note that we have to define conditions for equality and inequality separately, since there is the possibility that the relation between two objects is undetermined. This will be the case if the relation is neither explicitly equal nor unequal. Finally, we consider leaf values. Leaf values are equal if the nodes in the constraint net to which they refer are equal.

GeomStructure

GeomStructure

$$\forall\ g1,g2 : \mathrm{dom\ leaves} \bullet$$
$$\mathrm{leaves}(g1) \neq \mathrm{leaves}(g2) \Leftrightarrow \mathrm{unequal}(g1, g2) \wedge$$
$$\mathrm{leaves}(g1) = \mathrm{leaves}(g2) \Leftrightarrow \mathrm{equal}(g1, g2) \wedge$$
$$\mathrm{null}(g1) \wedge \mathrm{null}(g2) \Rightarrow \mathrm{equal}(g1, g2)$$

A ground geom is one which has no undefined parts. This will be useful when we come to consider operations which construct new geoms, since many of these will only be defined when their arguments are ground.

GeomStructure

GeomStructure

groundGeom _ : $\mathbb{P}$ GEOM

$$\mathrm{groundGeom} = \mathrm{dom}\ (\mathrm{parts} \rhd \mathrm{undefined}) \cup \mathrm{dom\ leaves}$$

We will now define some operations on the geom structure. Before we do so, we define some constraints on the before and after states which ensure that the geom structure can only change monotonically.

ΔGeomStructure

GeomStructure,

GeomStructure'

$$\mathrm{equal}' \supseteq \mathrm{equal} \wedge \mathrm{unequal}' \supseteq \mathrm{unequal} \wedge$$
$$\mathrm{labelledParts}' \supseteq \mathrm{labelledParts} \wedge \mathrm{null}' \supseteq \mathrm{null} \wedge \mathrm{leaves}' \supseteq \mathrm{leaves} \wedge$$
$$\mathrm{undefined} \supseteq \mathrm{undefined}' \wedge \mathrm{unused} \supseteq \mathrm{unused}'$$

The geom structure must be consistent both before and after the tell operation. Geoms that are equal (resp. unequal) before an operation are equal (resp. unequal) after it. The set of geoms with parts may be increased, but no geom which already has parts may gain new ones. Similarly, the set of node-valued

leaves may be increased, but once a geom is node-valued, it remains so. Notice that the sets of undefined and unused geoms always reduce.

We will define an infix relation $\_ <_{GS} \_$ which implies that one geom structure is entailed by another.

$$\_ <_{GS} \_ : GeomStructure \times GeomStructure$$

$$\begin{aligned}
&_ <_{GS} _ = \{a,b: GeomStructure \mid \\
&(a.equal \subseteq b.equal) \land (a.unequal \subseteq b.unequal) \land \\
&(a.parts \subseteq b.parts) \land (a.null \subseteq b.null) \land (a.leaves \subseteq b.leaves)\}
\end{aligned}$$

Any geoms which are equal (resp. unequal) in the first are equal (resp. unequal) in the second; the parts, null geoms and node valued geoms of the first are all contained in the second. In short, the second structure contains all information in the first. We will use this in the following schema to define a minimal change.First we define.

$$GeomPair \mathrel{\widehat=} [\, g1,g2: GEOM \,]$$

GeomEquality

$$\begin{aligned}
&\Delta GeomStructure \\
&GeomPair \\
&\hline
&equal'\ (g1, g2) \land \\
&\forall\ gs: GeomStructure \mid gs.equal\ (g1, g2) \bullet \\
&(\theta GeomStructure <_{GS} gs) \Rightarrow (\theta GeomStructure' <_{GS} gs)
\end{aligned}$$

This schema asserts that following a (successful) tell, the two geoms g1 and g2 will be equal. The second condition states that the change is minimal: the new relations of equality and inequality are the smallest possible to allow the two geoms to be equal. Any consistent structure which implies the initial state of the schema and the equality of the two input values also implies the final state of the geom. This ensures that other undefined geoms are not equated or made unequal and that no undefined geoms gain any parts other than through equating them with defined ones.

Clearly, the tell will always succeed if the two geoms are already equal, and

will always fail once the geoms have been made unequal, because of the monotonicity properties asserted in ΔGeomEquality.

In the notation we are developing we will put

g1 = g2

for geom equality.

We have not yet considered how new geoms may be added to a geom structure. We will therefore look now at constructor functions which create geoms. The geom constructor GeomUnion creates a new geom from two others. The new geom has the combined labels and parts of both constituents. Where two parts have the same label in both originals, then the two parts are equated. The operation fails if this is not possible. It suspends if either of the two geoms is undefined. We first define a function union which given any two geoms returns their union.

GeomStructure

GeomStructure

union : GEOM × GEOM × GEOM

---

union = {g1,g2,g3 :GEOM |
labelledParts g3 = labelledParts g1 ⊕ labelledParts g2 ∧
∀ l: LABEL | l ∈ (dom (labelledParts g1) ∩ dom (labelledParts g2)) •
equal (labelledParts g1 l, labelledParts g2 l) }

It is possible that there is more than one geom which satisfies the condition of being the union of the two given ones. In this case, one of them is chosen arbitrarily: to this extent, the definition is loose. However, it is easy to see that all geoms which are the union of two given ones will be equal. Notice also that the union is defined with respect to a given structure.

Now we can define the operation GeomUnion which succeeds if it is possible to

create a new geom which is the union of a given pair of geoms.

GeomUnion

ΔGeomStructure

GeomPair

g! : GEOM

---

union'(g1,g2, g!) $\wedge$ g! $\in$ dom labelledParts'

$\forall$ gs: GeomStructure | gs.union(g1,g2, g!) $\wedge$
g! $\in$ dom gs.labelledParts $\bullet$

(θGeomStructure $<_{GS}$ gs) $\Rightarrow$ (θGeomStructure' $<_{GS}$ gs)

We will define GeomDifference similarly. In this case, the difference of two geoms is a geom which has the labels and parts of the first geom wherever these do not occur in the second. Once again, for the operation to succeed we require that where the two original geoms have matching labels, then the parts should be equal.

GeomStructure

GeomStructure

difference : GEOM $\times$ GEOM $\times$ GEOM

---

difference = {g1,g2,g3 :GEOM |
labelledParts g3 = labelledParts g1 $\lhd$ dom labelledParts g2 $\wedge$
$\forall$ l: LABEL | l $\in$ (dom (labelledParts g1) $\cap$ dom (labelledParts g2)) $\bullet$
equal (labelledParts g1 l, labelledParts g2 l)}

The operation of GeomDifference succeeds if there is some geom in the after state of the operation which is the difference of the given geom pair.

GeomDifference

ΔGeomStructure

GeomPair

g! : GEOM

---

difference'(g1,g2,g!) $\wedge$ g! $\in$ dom labelledParts'

$\forall$ gs: GeomStructure | gs.difference(g1,g2,g!) $\wedge$
g! $\in$ dom gs.labelledParts' $\bullet$

(θGeomStructure $<_{GS}$ gs) $\Rightarrow$ (θGeomStructure' $<_{GS}$ gs)

Sometimes we will want to treat geoms just as bags of objects. In this case, the

labellings are irrelevant, but must be distinct within the structure. We will assume a function which generates a set of 'unused' labels for any geom structure and define it axiomatically as

$$\begin{array}{|l} \text{newLabel} : \text{GeomStructure} \rightarrow \mathbf{P}\ \text{LABEL} \\ \hline \\ \forall\ \text{gs:GeomStructure; g:GEOM} \mid \text{g} \in \text{dom gs.labelledparts} \bullet \\ \qquad \text{disjoint} < \text{newLabel gs, dom gs.labelledParts g} > \\ \\ \end{array}$$

Now we can define an operation which creates a 'bag-like' part labelling as follows

$$\begin{array}{l} \text{createBagGeom} \mathrel{\widehat{=}} [\Delta\text{GeomStructure; s:seq GEOM; g!:GEOM} \mid \\ \qquad \text{ran labelledParts' g!} = \text{ran s} \ \wedge \\ \qquad \text{dom labelledParts' g!} \subseteq \text{newLabel } \theta\text{GeomStructure}\ ] \end{array}$$

In the notation we will write such a geom as a list of geom names in curly brackets. e.g.

{a,b,c}

defines a geom with three parts. The labels are unspecified, but guaranteed to be unique in the structure. If we want to access one of the parts, we need to have an 'anonymous label'. So we define a function which returns any part of a geom. We need to be able to do this consistently returning the same part, so we will make the label returned dependent on the set of labels in the geom. Thus,

$$\begin{array}{|l} \text{anyLabel} : \ \mathbf{P}\ \text{LABEL} \rightarrow \text{LABEL} \\ \hline \\ \forall\ \text{labels:}\ \mathbf{P}\ \text{LABEL} \bullet \text{anyLabel labels} \in \text{labels} \\ \\ \end{array}$$

and we use the function to access parts of a geom as follows

$$\begin{array}{l} \text{anyPart} \mathrel{\widehat{=}} [\text{GeomStructure; g1: GEOM; g!:GEOM} \mid \\ \qquad \text{g!} = \text{labelledParts g1 (anyLabel (dom labelledParts g1))}] \end{array}$$

We will write this as

g._

Thus,

{a,b,c}._

might refer to a, b or c, but repeated applications always refer to the same element because the label chosen by anyLabel depends on the label set of the given geom.

This will be useful in the next chapter to implement distributed operations, which are applied to all parts of a geom, when we will pick any part, apply an operation to it and then repeat the process on the geom minus the given part. In order to achieve this we will need to be able to remove a part without knowing its label. The partsDifference of two geoms is equal to the first geom with all parts that are contained in the second removed. It differs from geomDifference in that we do not consider the labels of the second geom at all.

GeomStructure

GeomStructure

partsDifference : GEOM × GEOM × GEOM

---

partsDifference = {g1,g2,g3 :GEOM |
labelledParts g3 = labelledParts g1 ⊳ equal (ran labelledParts g2) }

The corresponding operation is

SetDifference

ΔGeomStructure

GeomPair

g! : GEOM

---

partsDifference'(g1,g2,g!) ∧ g! ∈ dom labelledParts'

∀ gs: GeomStructure | gs.setDifference(g1,g2,g!) ∧
g! ∈ dom gs.labelledParts' •

(θGeomStructure $<_{GS}$ gs) ⇒ (θGeomStructure' $<_{GS}$ gs)

In the notation, we will write union, difference and set difference as (respectively)

g1 = g2 + g3

g1 = g2 - g3

g1 = g2 \ g3

In distributed expressions we will combine anonymous labels and set difference along the lines of

```
doSomething(g) ::
        something(g._) , doSomething(g \ g._)
```

This is discussed in much more detail in Chapter 6, section 6.2.3.

We may use geoms to represent sequences by defining a one-to-one function

from integers to LABELS.

integerLabel: $\mathbb{N} \rightarrowtail$ LABEL

Now we can use this function to create a part labelling, given a sequence of geoms.

seqGeom == λ (s: seq GEOM • integerLabel^{-1} ; s)

We will write such a sequence as a comma-separated list in round brackets, i.e.

(g1, g2, ... , gn)

The operation createSeqGeom augments the structure with such a geom.

createSeqGeom $\widehat{=}$ [ΔGeomStructure; s:seq GEOM; g!:GEOM |
labelledParts' g! = seqGeom s]

Additionally we will allow the use of integers to reference parts of such geoms, e.g.

(a,b,c).1

is just a. That is, in the expression '1' denotes the label which is mapped onto the integer 1 by the injection integerLabel.

A geomMap is a relation which between three geoms with identical labels, in which the parts of the third are ordered pairs of the original parts. We will use this constructor to implement representations by defining mappings between domain model objects and interface model objects.

GeomStructure

GeomStructure

geomMap : GEOM × GEOM × GEOM

---

geomMap = {g1,g2,g3 :GEOM |
dom labelledParts g1 = dom labelledParts g2 = dom labelledParts g3 •
$\forall$ l: LABEL | l $\in$ dom (labelledParts g1) •
labelledParts (labelledParts g3 l) =
seqGeom < labelledParts g1 l, labelledParts g2 l > }

As before we define a corresponding operation.

GeomMapping

ΔGeomStructure
GeomPair
g! : GEOM

---

geomMap'(g1,g2,g!) $\wedge$
$\forall$ gs: GeomStructure | gs.geomMap(g1,g2,g!) $\bullet$
(θGeomStructure $<_{GS}$ gs) $\Rightarrow$ (θGeomStructure' $<_{GS}$ gs)

In our notation, we will write a geom mapping as

g = g1 @ g2

Thus, suppose that the labels and parts of g1 and g2 are

g1 = < a: part1, b:part2, c:part3 >
g2 = < a: part4, b:part5, c:part6 >

Then the effect of

g = g1 @ g2

is to make

g = < a: (part1,part4), b:(part2,part6), c:(part3,part7) >

Each geom is associated with a constraint net which contains the set of nodes which occur at its leaves. We can 'flatten' a geom to produce a labelling of the nodes in the constraint net by concatenating the labels of nested parts of the geom. That is to say, the parts of the flattened geom will all be leaf valued, and their labels will be the 'path' from the original geom to the leaf. Concatenation is associative and if two distinct labels are concatenated with the same label, the result in each case will be distinct. Thus

concatLabel: LABEL $\rightarrow$ LABEL $\rightarrow$ LABEL

$\forall$ a,b,c:LABEL $\bullet$
concatLabel (concatLabel a b) c = concatLabel a (concatLabel b c)
$\wedge$ concatLabel a b = concatLabel a c $\Leftrightarrow$ b = c
$\wedge$ concatLabel b a = concatLabel c a $\Leftrightarrow$ b = c

We define concatLabel as a curried function so that we can apply the arguments separately.

A flattened geom is a mapping from labels to leaf-valued geoms. The function flatten has the same signature as labelledParts, but note that we are not altering the geom structure by applying it. The schema just says that

- flatten is only defined for ground structures;
- if a labelled part of a geom is a leaf, then the flattened version of the geom maps that label to the leaf itself;
- if a labelled part of a geom also has parts, then the flattened version of the geom concatenates that label to the flattened version of the parts.

GeomStructure

GeomStructure

flatten: GEOM ⇸ LABEL ⇸ GEOM

---

dom flatten = groundGeom ∩ dom labelledParts ∧

∀ geom, part: GEOM; j,k:LABEL |
groundGeom(geom) ∧ part = labelledParts geom j •

part ∈ dom leaves ⇔ flatten geom j = part ∧

part ∈ dom labelledParts ⇔ flatten geom (concat j k) = flatten part k

We will use the symbol '#' to represent a flattened geom. Thus #g is the geom whose parts are all the leaves of g. This is a useful function because it enables us to check for identical structures. Iff two geoms have the same structure then the domains of their flattened versions are the same.

Now we can define a mapping between geoms by associating corresponding nodes in the flattened versions of the geoms. We will use this mapping in the next section when we define a constraint which asserts that one geom instantiates

another.

GeomStructure

GeomStructure

$$netMap : GEOM \times GEOM \rightarrow NodeMap$$

$$dom\ netMap = groundGeom \times groundGeom \wedge$$

$$\forall g1,g2: dom\ leaves \bullet netMap(g1,g2) = \{leaves\ g1 \mapsto leaves\ g2\}$$

$$\forall g1,g2: GEOM \mid dom\ flatten\ g1 = dom\ flatten\ g2 \bullet$$
$$netMap\ (g1,g2) = \{j: LABEL \mid j \in dom\ flatten\ g1 \bullet$$
$$leaves\ (flatten\ g1\ j) \mapsto leaves\ (flatten\ g2\ j)\ \}$$

The definition says that if the two geoms are both leaves, then the corresponding nodes in the constraint net are mapped, otherwise pairs of nodes with corresponding 'flattened' labels are mapped. The two constraints must have the same structure before they can be mapped. We check this by checking the domains of the flattened geoms.

5.4 The store

Now we can define the store and the operations on it. The store holds two kinds of information: information about structure is modelled as a geom structure and information about spatial relations between geoms as a constraint net.

$$Store \mathrel{\widehat=} [\ ConsistentNet;\ GeomStructure\ ]$$

The operation tell results in a (possibly) changed store and may have one of two outcomes: either it succeeds or it fails. An ask always leaves the store unchanged and the outcome may include suspension: it may not be possible to resolve the query with the information currently held in the store. If a tell fails, then the store is left unchanged.

$$RESULT ::= succeed \mid fail \mid suspend$$

$$ConstraintOp \mathrel{\widehat=} [\ \Delta Store;\ outcome! : RESULT\ ]$$

We require that a tell succeeds whenever possible.

$$Tell \mathrel{\widehat=} [\ ConstraintOp \mid$$
$$outcome! \in \{succeed, fail\} \wedge$$
$$outcome! = fail \Leftrightarrow \Xi Store \wedge$$
$$(\exists\ gs: ConstraintOp \bullet gs.outcome\ ! = succeed \wedge gs\ !\ Store = \theta Store\ )$$
$$\Rightarrow outcome! = succeed\ ]$$

$$Ask \mathrel{\widehat=} [\ ConstraintOp \mid \Xi Store\ ]$$

Geom constructors may change the store, but they can also suspend if the arguments are undefined. In fact, the constructors may be considered as a combination of ask and tell, but it will be simpler for us to model them as a special case.

AskTell ≙ [ConstraintOp |
outcome! = fail ⇔ ΞStore ∧
(∃ gs: ConstraintOp • gs.outcome ! = succeed ∧ gs ⍞ Store = θStore)
⇒ outcome! = succeed]

5.4.1 Ask and tell for spatial constraints

Now we will define ask and tell for spatial constraints. These operations have no effect on the geom structure. The operation tell succeeds if the constraint net being told can be merged with the store and it fails otherwise. In the former case, the merge is actually performed and the store is changed; in the latter, the store is unchanged.

SpatialTell ≙ [Tell; tell? : ConstraintNet |
ΞGeomStructure ∧
(outcome! = succeed) ⇔ θConsistentNet' = θConsistentNet + tell? ∧
(outcome! = fail) ⇔ (θConsistentNet,tell?) ∉ dom +]

The operation ask *succeeds* if a constraint net can be merged with the store, without adding new information; it *fails* if the constraintNet cannot be merged with the store; and it *suspends* if the net can be added to the store, but adds new information. The operation only checks the possibility of merging the net with the store, it does not perform the merge, so the store is left unchanged by an ask.

SpatialAsk ≙ [Ask; ask? : ConstraintNet |
(outcome! = succeed) ⇔ θConsistentNet = θConsistentNet + ask? ∧
(outcome! = fail) ⇔ (θConsistentNet,ask?) ∉ dom +]

Note that the definition of Ask ensures that when the outcome of an ask is neither succeed nor fail, then it is suspend. Therefore, we do not need to cover this case explicitly.

5.4.2 Ask and tell for geoms

The operations on the geom structure may be defined similarly. A successful GeomUnion is one in which the before and after state of the geom structure satisfies the condition of GeomUnion and the spatial constraints are unchanged. A GeomUnion suspends if either of the given geoms is undefined and it fails if it neither succeeds nor suspends.

TellGeomUnion ≙ [AskTell; GeomPair; g! |
(outcome! = succeed) ⇔ ΞConsistentNet ∧ GeomUnion ∧
(outcome! = suspend) ⇔ undefined(g1) ∨ undefined(g2)]

AskGeomUnion ≙ [Ask; GeomPair; g! |
(outcome! = succeed) ⇔ union(g1,g2,g!) ∧
(outcome! = suspend) ⇔ undefined(g1) ∨ undefined(g2)]

We will write ask and tell for geoms using '!' and '?'. Thus in our notation, we will write TellGeomUnion as

! g = g1 + g2

and AskGeomUnion as

? g = g1 + g2

Ask and tell of geom difference are defined similarly.

TellGeomDifference ≙ [AskTell; GeomPair; g! |
(outcome! = succeed) ⇔ ΞConsistentNet ∧ GeomDifference ∧
(outcome! = suspend) ⇔ undefined(g1) ∨ undefined(g2)]

AskGeomDifference ≙ [Ask; GeomPair; g! |
(outcome! = succeed) ⇔ difference(g1,g2,g!) ∧
(outcome! = suspend) ⇔ undefined(g1) ∨ undefined(g2)]

In the case of SetDifference, the constructor suspends if it is not possible to determine whether the parts of the second geom are equal to those of the first.

TellSetDifference ≙ [AskTell; GeomPair; g! |
(outcome! = succeed) ⇔ ΞConsistentNet ∧ SetDifference ∧
(outcome! = suspend) ⇔
(ran labelledParts g1 × ran labelledParts g2) ⊆ (equal ∪ unequal)]

AskSetDifference ≙ [Ask; GeomPair; g! |
(outcome! = succeed) ⇔ partsDifference(g1,g2,g!) ∧
(outcome! = suspend) ⇔
(ran labelledParts g1 × ran labelledParts g2) ⊆ (equal ∪ unequal)]

A GeomMapping can be told if at least one of its nodes is defined. If it is a node with parts, then the labels of the other two nodes will be the same. If it is a leaf node, then the relation geomMap will not hold and the operation will fail.

TellGeomMapping ≙ [AskTell; GeomPair; g! |
(outcome! = succeed) ⇔ ΞConsistentNet ∧ GeomMapping ∧
(outcome! = suspend) ⇔
undefined(g1) ∧ undefined(g2) ∧ undefined(g3)]

AskGeomMapping ≙ [Ask; GeomPair; g! |
(outcome! = succeed) ⇔ geomMap(g1,g2,g!) ∧
(outcome! = suspend) ⇔
undefined(g1) ∧ undefined(g2) ∧ undefined(g3)]

This means that so long as one of the three has a defined label set, the mapping can be told (although it might then fail). For example,

! g1 = <a,b,c> @ g2

will give g1 and g2 the label set <a,b,c>, or fail if either already has a differently defined set of labels.

In the case of geom constructors a tell may suspend until the parts of a geom are known. That is to say, there is an *implicit ask* in constructors. Constructors may alter the store by creating new geoms, and in the process information may be added concerning their arguments. In the case of GeomUnion for example, following the construction of the union, parts of the arguments may have been equated. By contrast, constraints on geoms do not cause the creation of new geoms, nor does tell suspend. Ask and tell for geom equality are defined as follows.

TellGeomEquality ≙ [Tell; GeomPair |
(outcome! = succeed) ⇔ ΞConsistentNet ∧ GeomEquality ∧
(outcome! = fail) ⇔ unequal(g1,g2)]

AskGeomEquality ≙ [Ask; GeomPair |
(outcome! = succeed) ⇔ equal(g1,g2) ∧
(outcome! = fail) ⇔ unequal(g1,g2)]

5.4.3 Instantiations

We will say that one geom instantiates another if the two have the same structure and the parts of the constraint net associated with the first instantiate those of the second.

Store

Store

geomInstantiate : GEOM × GEOM

---

geomInstantiate = { g1,g2: GEOM | (g1,g2) ⊆ dom netMap ∧
instantiates(netMap(g1,g2), θStore, θStore)}

Instantiations obviously figure strongly in our exposition: we will need to be able to say that one geom instantiates another in order to model the relation between display states and the interface state. Therefore we require a constraint which as-

serts that one geom instantiates another.

GeomInstantiation

ΔGeomStore

GeomPair

---

geomInstantiate'(g1,g2) ∧

∀ gs: GeomStore | gs.geomInstantiate(g1,g2) •

(θGeomStructure $<_{GS}$ gs ↾ GeomStructure) ⇒
(θGeomStructure' $<_{GS}$ gs ↾ GeomStructure) ∧

(θConsistentNet $<_{CN}$ gs ↾ ConsistentNet) ⇒
(θConsistentNet' $<_{CN}$ gs ↾ ConsistentNet)

As in the previous section, we use the entailment relations $<_{GS}$ and $<_{CN}$ to assert that the changes are minimal. The result of telling the constraint may be to alter the constraint net, or to update the geom structure. The operation suspends until the first geom is ground.

TellGeomInstantiation ≙ [AskTell; GeomPair |
(outcome! = succeed) ⇔ GeomInstantiation ∧
(outcome! = suspend) ⇔ groundGeom(g1)]

AskGeomInstantiation ≙ [Ask; GeomPair |
(outcome! = succeed) ⇔ geomInstantiate(g1,g2) ∧
(outcome! = suspend) ⇔ groundGeom(g1)]

In the notation we will write

g1 << g2

to say that geom g1 instantiates geom g2.

5.4.4 Stability of the constructors

Constraints that may be applied to the geom structure are equality and instantiate. These may be asked, in which case the constraint succeeds, suspends or fails, or they may be told when the constraint either succeeds or fails. Operations on the geom structure also include constructors. These may suspend, succeed by creating a new node or fail. We have defined constructors union, difference, setDifference and geomMap. Both constructors and constraints are stable. Once a constraint has been successfully told in a store, it can always be told; once it has failed it will always fail. With constructors, the stability property implies that once a constructor has succeeded, creating a given geom, then it will always

succeed, creating a geom which is indistinguishable from that created previously. Note that this is not the same as saying that the result will always be the same: it might be that the result from the first constructor gets modified by constraints told subsequently. What we are saying is that the result of a constructor is always equal to the result from previous applications of the constructor, *including any changes which have been caused by subsequent constraint tells, either to the previous result or to the arguments.*

For example, suppose we have the four constraints

! g1 = <a:A, b:B>,

! g2 = <a:A,c:C>,

! g3 = g1 + g2,

! g1.a = (A,B)

The stability property requires that

- if the set of tells is successful, then it will be successful *for any allowed order of telling;*
- if the set of tells is successful, the final state of the store is the same, *for any allowed order of telling;*

Given a set of successful constraint tells and constructor applications, any sequence of application produces the same result.

In our example, the constructor union suspends until both g1 and g2 are ground. The label reference in g1.a = (A,B) suspends until the labels of g1 are known. However, the third and fourth statements may be executed in either order. If the union is executed first, then it will result in g1.a and g2.a being equated since union equates parts with the same label. Thus, when the fourth statement is executed, both g1 and g2 will have part a equal to (A,B). Conversely, if the fourth statement is executed first, when g1.a and g2.a are equated, g1.a will already have the value (A,B) and following the equation, so will g2.a.

5.5 Agents

We can now describe the behaviour of simple agents communicating through the store. An agent consists of a *basic action* and a replacement. The basic action consists of an ask–tell pair and a *guard*. For the moment we defer the discussion of the guard. The ask and tell constraints may apply to the geom structure, or to its associated constraint net.

Agents are executed with a binding which identifies a sub net of the store. Informally, an agent fails if either its ask or its tell fail, it suspends if the ask

suspends and it succeeds if both ask and tell succeed. The system at any moment consists of two parts: these are a set of agents awaiting evaluation and the current store. Basic actions may be either ask-tell pairs, or *guards*. A guard is an agent which acts as a basic action, committing to the store successfully, or leaving it unchanged in the case of failure.

The cycle of evaluation consists of finding an agent whose basic actions can be successfully evaluated, updating the store with the tell constraint, and adding the replacement agent to the set of agents awaiting evaluation. The choice of agent to be selected for evaluation is indeterminate. Once an agent has been selected and evaluates with failure, it is removed from the set of agents awaiting evaluation. The monotonic nature of the constraint system means that if a set of agents succeed (i.e. all succeed) for some order of evaluation, then it will succeed for any order of evaluation. Similarly, once an ask has succeeded in the store, it will always succeed. The basic actions are performed atomically: that is to say, if a basic action fails, there is no change to the store. Each agent represents a thread of evaluation consisting of itself and its descendant replacements, which ultimately terminates in success or failure. The execution of an agent is not, in general, atomic since it may fail having added constraints to the store.

Later on, we will consider three operations to form composite agents: disjunction, conjunction and parallel composition. However, for the moment we will consider only simple agents. Our starting point will be the set of syntactically valid expressions in the notation that we will develop in this chapter and a set of names to be used in the expressions. We will develop the concrete syntax for the notation in the next section. In this section we consider only agents defined in terms of specific nodes in the store: parameters and other variables all refer to individual nodes.

Syntactically, a *program* is a set of agent definitions.

program :: agentDefinition $^{+}$

agentDefinition :: goal :: body .

body :: [ConstraintExpression] [ConstraintExpression] goal

goal :: agentName ([var [, var]])

In order to derive constraints from expressions, we will need a meaning function, M_C which allows us to derive constraints from expressions. M_C maps expressions onto a function which maps bindings to constraints. A binding is just a sequence of geoms. Expressions which represent goals will also have an

argument list, which is a list of indices into the binding. All agents have a binding.

Binding == seq GEOM

ArgList == seq $\mathbb{N}$

M_C : EXPR ⇸ Binding ⇸ ConstraintNet

Now we can define the set of valid constraint expressions as

ConstraintExpression == dom M_C

An agent may be a simple agent consisting of a basic action and a replacement or, recursively, some composition of agents. The replacement for the agent may be a goal with arguments, which is to be replaced by the corresponding clause body in the program, or it may be some other agent.

Compound agents are

- disjunct
- parallel
- sequence

Thus we may define

AGENT ::= basicAction << EXPR × EXPR × Binding >> |
guard <<AGENT × Binding>> |
disjunct << AGENT × AGENT × AGENT × Binding >> |
parallel <<AGENT × AGENT × Binding >> |
sequence <<AGENT × AGENT × Binding >> |
goal << NAME × ArgList × Binding >> |
succeed |
fail

A parallel composition of agents fails if either of its components fail and suspends if any are suspended, but succeeds only when all have succeeded. A sequential composition is similar to a parallel one, except that the second component is not evaluated until the first has succeeded. If a sequential composition succeeds (resp. fails) then a parallel composition with the same components must succeed (resp. fail). However, it is possible for a sequential composition to suspend where the equivalent parallel composition would terminate.

It will be useful to access the bindings of agents, so we define a function for

doing this.

$$\begin{array}{|l} \text{binding: Agent} \rightarrow \text{Binding} \\ \hline \\ \forall\ a, a1,a2,a3\text{: AGENT; b: Binding; e1,e2: EXPR; n:NAME; arg: ArgList} \mid \\ \qquad \text{basicAction(e1,e2,b)} = a \ \vee\ \text{disjunct (a1,a2,a3,b)} = a \ \vee \\ \qquad \text{parallel(a1,a2,b)} = a \vee \text{sequence(a1,a2,b)} = a \vee \text{goal(n,args,b)} = a \bullet \\ \qquad \text{binding(a)} = b \ \wedge\ \text{binding(guard(a))} = b \\ \\ \end{array}$$

The function args returns the bound arguments of a goal agent. That is to say, it is the binding obtained by replacing the indices in the argument list of the goal with their associated bindings.

$$\begin{array}{l} \text{args: goal} \rightarrow \text{Binding} == \forall\ g\text{: AGENT; b: Binding; n:NAME; a: ArgList} \mid \\ \qquad \text{goal(n,a,b)} = g \bullet\ \text{args(g)} = a \upharpoonright b \end{array}$$

Syntactically, the definition of a goal is given by the body of the clause whose head matches that of the goal. *Goal reduction* occurs when a goal is looked up in the program and replaced by the corresponding definition. The syntactic definition of a program does not take account of the binding in use at the time of goal reduction. That is to say, syntactically, a program is a function from names to definitions. Strictly speaking, we should define program as

$$\text{program}_0 : \text{NAME} \nrightarrow \text{UnboundAgent}$$

where UnboundAgent has a recursive definition with the same structure as AGENT, but without the bindings. We could then define

$$\text{boundAgent: (UnboundAgent} \times \text{Binding)} \nrightarrow \text{AGENT}$$

This would have the advantage of making explicit the relationship between the syntactic objects (goal names and definitions) and their semantic counterparts. However, the additional definitions required will only be used in this context, so we will be economical and include look-up and binding within the same function, making program be a function from goals to agents. The agent returned by the function will have a binding which includes the argument list of the goal, but which

additionally has other (as yet unspecified) local arguments.

Program

$$
\begin{array}{l}
\text{program: AGENT} \nrightarrow \text{AGENT} \\
\hline
\text{dom program} \subseteq \text{ran goal} \ \wedge \\
\forall\ \text{a: AGENT} \mid \text{a} \subseteq \text{dom program} \bullet \text{args(a)} \subseteq \text{binding(program a)} \ \wedge \\
\forall\ \text{a, a1,a2,g: AGENT; b: Binding} \mid \text{a} \in \text{ran program} \wedge \text{g} \in \text{ran goal} \bullet \\
\text{a = disjunct (a1,g,a2,b)} \ \vee\ \text{a = disjunct (a1,a2,g,b)} \ \vee \\
\text{a = parallel (a1,g,b)} \ \vee\ \text{a = parallel (g,a1,b)} \ \vee \\
\text{a = sequence (a1,g,b)} \ \vee\ \text{a = sequence (g,a1,b)} \\
\Rightarrow \text{g} \in \text{dom program}
\end{array}
$$

In a syntactically valid program, clause bodies are disjunct, parallel or sequential compositions of agents. The second universally quantified predicate in the schema ensures that every goal which occurs in a clause body must be defined in the program.

A *configuration* consists of a program, a store and an agent.

$$\text{Configuration} \ \widehat{=}\ [\text{Program; store: Store; agent :AGENT }]$$

Computation proceeds by a process of *reduction* in which the agent is evaluated and replaced. The rules for replacement depend on the kind of expression being reduced.

We will now define some operations which describe reductions of a configuration. The following two definitions will be useful.

$$\text{ChangedAgent} \ \widehat{=}\ \Xi\text{Configuration} \setminus (\text{agent, agent'}) \ \wedge\ \Delta\text{Configuration}$$

$$\text{ChangedStore} \ \widehat{=}\ \text{ChangedAgent} \setminus (\text{store,store'}) \wedge \Delta\text{Configuration}$$

The configuration ChangedAgent is one in which only the agent has changed, and ChangedStore is one in which both agent and store have changed.

A configuration reduces by performing a basic action, and updating the store, or by expanding a goal into the corresponding clause body in the program. Following a reduction, both the agent in the configuration and possibly the store will have changed. The reduced form of a basic action is either succeed, or fail, or the agent itself (in the case of suspension) and the reduced form of a goal is the clause body associated with that goal in the program, together with an updated binding.

Now we define a relation derivation whose elements are all pairs of configurations such that there is a series of reductions from one to another. We will leave it loose for the moment.

derivation: Configuration ↔ Configuration

---

derivation = derivation$^+$ ∧
∀ a,b:Configuration • derivation(a,b) ⇔ ¬ derivation(b,a)

The relation is transitive, so that if there if there is a derivation from configuration c1 to c2 and from c2 to c3, then there is a derivation from c1 to c3. The second constraint of the predicate ensures that there are no cyclic derivations.

A direct derivation is one for which there are no intermediate steps.

directDerivation: Configuration ↔ Configuration

∀ a,b,c:Configuration •
derivation(a,b) ∧ derivation(b,c) ⇔ ¬ directDerivation(a,c)

We will now define the operation of reduction. A configuration consists of an agent, a program and a store. When a transition occurs, some agent which is a descendant of the agent in the configuration is (recursively) reduced with the current program and store. This will yield a new program, store and agent (the old agent in which the reduced agent has been replaced by its reduction). We will define the reductions by cases, showing how the new agent and the reduced agent are related for each of the agent types. The following definition will be useful.

Transition

ΔConfiguration

ΔConfiguration$_R$

---

agent$_R \neq$ succeed $\wedge$

agent$_R \neq$ fail $\wedge$

θ Configuration $\neq$ θ Configuration' $\wedge$

θ ΔConfiguration \ (agent,agent') = θ ΔConfiguration$_R$ \ (agent$_R$,agent$_R$ ') $\wedge$

derivation(θ Configuration, θ Configuration') $\wedge$

derivation(θ Configuration$_R$, θ Configuration$_R$')

This schema says the following things:

- the constants succeed and fail can never be reduced;
- the current configuration is always changed by a transition;
- the reducing configuration (ΔConfiguration$_R$) is the current configuration, with the agent which is to be reduced;
- in any transition, there is a derivation between the before and after states of the current configuration and those of the reducing configuration.

So far, we have placed no restrictions on the values of the reduced agent and the agent in the current context. In fact the property of Transition is trivially satisfied by allowing the two to be identical. We will need later on to define additional properties which define which agent may be reduced in a given context. However, we start off with two forms of reduction in which the agent in the current context is reduced directly.

Basic reduction takes a basic action and translates the associated constraint expressions to constraint nets, which are then merged into the store using the operations ask and tell. If it succeeds, then the basic action is replaced by the agent succeed and the agent in the configuration is updated. If it fails, the replacement and updating are carried out, but the store will not have changed. If the basic action suspends, no change occurs in the configuration, so this case is disallowed by the property of transition (i.e. if an agent suspends, it does not reduce). In the case of basic reduction, there is no reducing agent, because ask and

tell are atomic. The new configuration is a direct derivation of the original.

BasicReduction

Transition

AskAndTell

$\exists$ e1, e2: ConstraintExpression; b:Binding •

agent = basicAction (e1,e2,b) $\wedge$ ask? = M_C b e1 $\wedge$ tell? = M_C b e2 $\wedge$

Succeed $\Leftrightarrow$ ChangedStore $\wedge$ agent' = succeed $\wedge$

Fail $\Leftrightarrow$ ChangedAgent $\wedge$ agent' = fail $\wedge$

directDerivation(θ Configuration, θ Configuration')

In a goal reduction, new nodes may be created in the store for the local variables of the clause body associated with the goal. Variables in the body of a clause which do not occur in the head are local and get instantiated with new node values when the goal is reduced. The current configuration is then updated by replacing the goal with the clause body.

Whereas basic reduction adds constraints to preexisting nodes in a network, goal reduction adds new, unconstrained, nodes to the network. As in the case of basic reduction, goal reduction does not involve the reduction of another agent, so the values of $agent_R$ do not need to be considered.

GoalReduction

Transition

AugmentNet

agent' = program agent $\wedge$

n? = #binding(program agent) - #args(agent) $\wedge$

ran binding agent' = newNodes! $\cup$ ran args(agent) $\wedge$

store = θ ConstraintNet $\wedge$ store' = θ ConstraintNet' $\wedge$

directDerivation(θ Configuration, θ Configuration')

This gives us derivations for configurations whose agent is either a basic action or a goal. We will now proceed by cases to consider derivations of configurations whose agent is neither a basic action nor a goal. In these cases, the agent in the current context is composite and reduction occurs by reducing one of its components.

A disjunct is a choice point where one of two replacement agents is chosen depending on the success or failure of a basic action (or guard).

DisjunctReduction

Transition

$\exists$ ba, a1,a2:AGENT; b:Binding •
agent = disjunct(ba,a1,a2,b) $\wedge$
ba = succeed $\Leftrightarrow$ agent' = a1 $\wedge$
ba = fail $\Leftrightarrow$ agent' = a2 $\wedge$
ba = $agent_R$ $\Leftrightarrow$ agent' = disjunct($agent_R$',a1,a2,b)

There are three cases: the choice agent may be succeed, fail or some basic action. In the case of succeed, the first of the two replacement agents is chosen; in the case of fail, the second is chosen; otherwise, there is a transition iff the choice agent has a transition in the current context and the replacement is a disjunct with the reduced choice agent. If there is no derivation for the basic action in the current configuration, then there is no derivation for the disjunct (i.e. it suspends).

A parallel composition has derivations which are the parallel composition of the derivations of its agents.

ParallelReduction

Transition

$\exists$ a1,a2:AGENT; b:Binding •
agent = parallel(a1,a2,b) $\wedge$
a1 = succeed $\Leftrightarrow$ agent' = a2 $\wedge$
a1 = fail $\Leftrightarrow$ agent' = fail $\wedge$
a2 = succeed $\Leftrightarrow$ agent' = a1 $\wedge$
a2 = fail $\Leftrightarrow$ agent' = fail $\wedge$
a1 = $agent_R$ $\Leftrightarrow$ agent' = parallel($agent_R$',a2,b) $\wedge$
a2 = $agent_R$ $\Leftrightarrow$ agent' = parallel(a1,$agent_R$',b)

If either component is succeed, then the composition reduces to the other component. If either is fail, it reduces to fail. Otherwise, if either component has a reduction, then the composition reduces to a new parallel composition with one of the agents replaced by its reduction. If neither agent has a derivation, then

neither does the parallel composition.

Finally, we complete the analysis by cases (excluding guards) by defining the derivations of a sequence. A sequence of agents fails if the first agent fails. It succeeds if both succeed. If the first agent of a sequence has a derivation in which it is reduced to some other agent (say r) then there is a derivation from the original sequence to a new sequence in which the first agent has been replaced by r.

SequentialReduction
Transition

$\exists$ a1,a2:AGENT; b:Binding •
agent = sequence(a1,a2,b) $\wedge$
a1 = succeed $\Leftrightarrow$ agent' = a2 $\wedge$
a1 = fail $\Leftrightarrow$ agent' = fail $\wedge$
a1 = $agent_R$ $\Leftrightarrow$ agent' = sequence($agent_R$',a2,b)

If the first agent has no derivation, then neither does the sequential composition.

A reduction is one of basic, goal, disjunct, parallel or sequential reductions (we will consider guard reduction later on).

Reduction $\widehat{=}$ BasicReduction $\vee$ GoalReduction $\vee$ DisjunctReduction $\vee$ ParallelReduction $\vee$ SequentialReduction

It will be useful for the definition of guards to consider the effects of a sequence of reductions: in particular we will need to know about terminating sequences of reductions. We can now define some useful sets of configurations in terms of their derivations. A terminalConfiguration is one which has no derivations.

terminalConfiguration = {Configuration | θ Configuration $\notin$ dom derivation • θ Configuration }

A successful (resp. failed) termination is a (terminal) configuration whose agent is succeed (resp. fail).

SuccessfulTerminations = {Configuration |
(θ Configuration).agent = succeed • θ Configuration }

FailedTerminations = {Configuration |
(θ Configuration).agent = fail • θ Configuration }

Terminations = SuccessfulTerminations $\cup$ FailedTerminations

A successfulDerivation of a configuration is a derivation which is a successful termi-

nation. Likewise, failedDerivation.

successfulDerivation = derivation ▷ SuccessfulTerminations

failedDerivation = derivation ▷ FailedTerminations

Our definition of reduction is not yet complete, because we have not considered guards. Before doing this, we need to consider the concept of commitment.

5.5.1 A note on commitment

Saraswat considers two basic forms of commitment operation to be applied when a disjunction of agents is evaluated. These are termed *Don't Know* and *Don't Care* commitment. In Saraswat's treatment of disjunction, each disjunct element consists of a basic action and a replacement to be used if the basic action succeeds. When a configuration reduces, it is possible that there may be more than one candidate for reduction in any given disjunction. This will be the case if two or more of the basic actions can successfully commit to the store. The difference between the two forms of commitment is in the way that such choice points are treated.

With Don't Care commitment, when a disjunction of agents is evaluated, if there is a basic action that succeeds, then the corresponding replacement agent is added to the set of active agents, and all the other branches of the disjunction are thrown away. If there is more than one candidate branch, then one of them is selected and the others discarded. Notice that there is no backtracking: the evaluation commits itself fully to the agent whose basic actions first evaluate successfully. If the replacement agent should fail following commitment, there is no return to the remaining branches of the disjunction (the *failure residue*) in order to try the other choices.

With Don't Know commitment, when a choice point is encountered two copies of the current configuration and two parallel threads of evaluation are created. One has the disjunction replaced by the replacement agent of the successful disjunct; the other pursues the failure residue, where the disjunction has been replaced by another disjunction which contains all the clauses of the original, except the one pursued in the successful branch. The failure residue may subsequently be split to pursue each of the remaining branches of the disjunction. When a configuration is split, the two copies are pursued entirely independently of each other, so that any constraints that get told in one thread of evaluation are unavailable to the other. As soon as one of the two configurations has terminated

successfully (i.e. its agent is succeed), then the evaluation commits to that configuration and the other is discarded.

The essential difference between Don't Know and Don't Care commitment is that with the former, we don't know in advance which of the candidates for replacement agent will be successful or not so we delay the choice between them until one has terminated successfully, whereas with the latter, we don't care whether the replacement can terminate successfully and make a commitment to one of the candidate replacement agents before any of them has been evaluated.

Thus, with Don't Know commitment, we are guaranteed that if there are replacement agents that *can* be evaluated successfully, then one of them *will* be evaluated. We do not have the same guarantee with Don't Care commitment, because we do not wait to discover the result of evaluating the replacement before committing to the thread of evaluation. With Don't Care commitment, it is possible for a disjunction to fail, even though there is a branch that could have completed successfully.

The implementation costs of Don't Know commitment are obviously higher than those of Don't Care commitment, since it is necessary to follow two configurations every time a choice point is encountered. When disjunctions are nested, it is possible for each independent thread of evaluation to split again, so that the number of independent threads being pursued may grow exponentially. This cost is not associated with Don't Care commitment because we only ever pursue one thread of execution.

Saraswat observes that in many cases it is possible to predict in advance which branches of a disjunction can succeed. Sometimes, we can guarantee that at most one of the basic actions of a disjunction can succeed. In this case, Don't Know and Don't Care commitment are observationally equivalent since we can be sure that Don't Know will succeed (fail) exactly when Don't Care succeeds (fails). This form of Don't Care commitment is termed *determinate*. Determinate commitment is a way to achieve the effect of Don't Know commitment without the associated implementation costs.

The form of commitment which we use is determinate: at most one branch of the disjunction can commit. The general form of disjunct expression which we use is

$$b \rightarrow a_1 \mid a_2$$

where b is a basic action consisting of the ask-tell pair

$ask(b_{ask})\ tell(b_{tell})$

Operationally, this commits to a_1 if b succeeds and to a_2 if b fails. In the cc() framework this disjunction would be expressed as

$ask(b_{ask})\ tell(b_{tell})\ \%\ a_1$

$ask(b_{ask}\sim)\ \%\ a_2$

$ask(b_{tell}\sim)\ \%\ a_2$

where the symbol '%' is a commitment operation, either Don't Know or Don't Care and where $ask(b_{ask}\sim)$ is an ask which succeeds (resp. fails,suspends) exactly when $ask(b_{ask})$ fails (resp. succeeds, suspends) and where $ask(b_{tell}\sim)$ is an ask which succeeds exactly when $tell(b_{tell})$ fails and otherwise either fails or suspends. The disjunction is determinate because there are no conditions under which there is a choice between a_1 and a_2. Whether we choose Don't Know or Don't Care commitment makes no difference to the outcome of evaluating the disjunction.

One way to define the unary operator ~ is to take a constraint net and return a net with the same nodes, but which has every label allowed in the original forbidden in the new, and every label forbidden in the original unknown in the new. So

$$\_\sim : ConstraintNet \rightarrow ConstraintNet$$

$\forall$ c: ConstraintNet; a:Arc •

$\sim c.unknown(a) = c.forbidden(a) \wedge$

$\sim c.forbidden(a) = c.allowed(a)$

There is, of course no guarantee that such a net is consistent. However, the following rules relating c and c~ follow directly from the observation that the instantiations of c and c~ must be disjoint since the allowed labellings of the nets are disjoint.

$ask(c) = fail \Rightarrow ask(c\sim) = succeed$

$ask(c) = suspend \Rightarrow ask(c\sim) = suspend$

$ask(c) = succeed \Rightarrow ask(c\sim) = fail$

$tell(c) = succeed \Rightarrow ask(c\sim) = fail, suspend$

$tell(c) = fail \Rightarrow ask(c\sim) = succeed$

In any configuration where ask(c) fails, ask(c~) will succeed. The rule for tell(c) means that in any configuration where tell(c) *would* succeed, ask(c) will fail or suspend.

Thus, to summarize, the form of disjunction which we are using is a special case of that proposed in the cc() framework for concurrent constraint programming. It is determinate, because at most one branch of the disjunction can succeed, therefore Don't Know commit can be implemented as Don't Care commit.

5.5.2 Guards

In the development so far, only ask and tell are allowed as basic actions. Telling a constraint either succeeds and changes the store, or fails leaving the store unchanged. However, as we have seen in the previous section, there is no such guarantee with other forms of agent, which may successfully tell some constraints before failing. The idea of guards is to allow arbitrary agents to be used as basic actions, so that in the event of failure, the store is left unchanged. The concept is similar to that of transactions in database systems, which can be rolled back in the event of failure.

Guards are observationally equivalent to atomic actions: the behaviour of a guard should be the same whether it is evaluated interleaved with other agents, or without interleaving. This implies that any constraints which get told during the execution of the guard should not be *published* (made available to other agents) until the guard has terminated successfully. Provided that publication of the constraints is atomic, the guard can be executed in an interleaved fashion, without this being apparent to other agents. Effectively, the guard appears to suspend until all of its basic actions can be evaluated in a single go.

We can now augment the definition of Reduction to include guard reduction.

GuardReduction

Transition

$$
\begin{aligned}
&((\text{agent} = \text{guard}(\text{succeed}) \wedge \text{agent}' = \text{succeed}) \vee \\
&(\text{agent} = \text{guard}(\text{fail}) \wedge \text{agent}' = \text{fail}) \vee \\
&(\text{agent} = \text{guard}(\text{agent}_R) \wedge \text{agent}' = \text{succeed} \wedge \\
&\qquad \text{successfulDerivation}(\theta\ \text{Configuration}_R, \theta\ \text{Configuration}_R')) \vee \\
&(\text{agent} = \text{guard}(\text{agent}_R) \wedge \text{agent}' = \text{fail} \wedge \\
&\qquad \text{failedDerivation}(\theta\ \text{Configuration}_R, \theta\ \text{Configuration}_R'))) \wedge \\
&\text{directDerivation}(\theta\ \text{Configuration}, \theta\ \text{Configuration}')
\end{aligned}
$$

The first two conditions state that if the guarded agent is succeed (resp. fail) , then the guard can reduce to succeed (resp. fail). The next two state that if the agent in the guard has a successful (resp.failed) derivation from the current configuration, then there is a reduction in which the guard is reduced to succeed (resp. fail). The final condition states that the reduction is atomic.

The essence of guard reduction is that the reductions of the guard are performed sequentially, without interleaving them with reductions from agents outside the guard.

Finally, we redefine the schema Reduction as

$\text{Reduction} \triangleq \text{Reduction} \vee \text{GuardReduction}$

5.5.3 Communication with the functional core

The cc() model as proposed by Saraswat is a model of concurrent computation. As such, it is intended that languages within the cc() framework be complete programming languages. Our intention is merely to model the spatial properties of the user interface, so we do not need to include other constructs commonly found in 'full' programming languages, such as the ability to compute with real numbers, or to perform operations on strings or to open and close external files or whatever. However, such abilities are useful or even necessary for modelling the functional core of the application. We need to provide some means for synchronization between the interface model, expressed using spatial constraint and external processes in the functional core.

We therefore introduce the idea of an *external* goal, which syntactically has the

same form as a goal, but for which there is no definition in the program. These behave as guarded agents, suspending until they are resolved by the functional core. We require external goals to be stable: once an external goal has resolved successfully (resp. failed), then it must be guaranteed that it will always resolve successfully (resp. fail). This is the only restriction which we place upon the functional core.

ExternalGoalStore ≙ [failures,successes : **P** AGENT |
disjoint<failures, successes>]

ΔExternalGoalStore ≙ [ΔExternalGoalStore |
failures' ⊇ failures ∧ successes ⊇ successes']

ExternalGoalReduction

failures,successes : **P** AGENT

---

disjoint<failures, successes>

5.6 Summary of the notation

In this section, we give an informal summary of the notation we have developed so far. A full definition of the syntax is given in Appendix B.

5.6.1 Ask and Tell

Constraints may be ask, tell or geom constructors. These latter are effectively basic actions consisting of an ask and a tell. We denote tell and constructor expressions with '!' and ask expressions with '?'. Thus,

! a = b , ? c = b

tells that a equals b and then asks whether b equals c.

5.6.2 Spatial constraints

Spatial constraints are lists of relation names in curly brackets, followed by a pair of leaf geoms. These may either be orientation or topological relations. The list can be qualified as required (abbreviated by the keyword **req**) or forbidden (**not**). For simplicity, we adopt the convention that where no orientation (resp. topological) relation is specified then all orientation (resp. topological) relations are allowed. The constraint will fail if the named geoms are not leaves. If the geoms are undefined, they will be bound to new nodes in the constraint net.

Thus, to say that two regions are not disjoint, we could say

! not {disjoint} (a,b)

In order to say that either they are disjoint, and one is vertically aligned with the other, we would say

! {disjoint, vAlign} (a,b)

5.6.3 Geom structure

We have defined three ways to represent labellings for geoms. We may define an arbitrary label set by putting a list of labels in angle brackets. For example,

! arrow = <head, tail, body>

defines arrow to be a geom with three parts labelled head, tail and body. The order is not important, so that

! arrow = < head, body, tail>

is equivalent. We may reference individual parts of the geom using a dot and the label. For example,

? {upon} (arrow.head, b}

asks whether the part of geom arrow labelled head is constrained to be upon geom b. As a shorthand we allow the use of

label : geom

pairs in structure definitions, which is equivalent to an equality constraint. For example,

? arrow = <head:a, body:b, tail:c>

is equivalent to

? arrow = < head, body, tail>, ? arrow.head = a, ? arrow.body = b, ? arrow.tail = c

We may create integer-labelled geoms by using a list in round brackets. So

! g = (a,b,c)

is equivalent to

g = <1:a, 2:b, 3:c>

In order to model bags of objects with distinguished, but anonymous, labels we use curly brackets.

! g = {a,b,c}

This asserts that g has the three components a, b and c and that each has a distinguished label. The difference between sets and sequences is shown by union. Thus,

! g = (a,b,c) + (d,e,f)

fails if a and d, or b and e, or c and f cannot be equated. Following a successful tell,

g will have three components. However, if

```
! g = {a,b,c} + {d,e,f}
```

succeeds then g will have six components.

In order to reference labels anonymously we use '._'. Thus,

```
g._
```

refers to some part of g. An arbitrary choice is made as to which part is chosen, but the definition of '._' guarantees that the same part is referenced on every occasion of use.

5.6.4 Constraints on geoms

Constraints on geoms are equality and instantiation. We have already discussed the latter in some detail. In the next chapter we will see how the instantiation constraint may be used to express rigid motion. Instantiation is written as

```
! a << b
```

which asserts that geom a is an instantiation of b: i.e. they have identical structure and the constraint net (leaf nodes) of a is an instantiation of the constraint net of b. Note that this suspends until a is ground: if the ground form of a has more than one required label on any arc, then the constraint fails because an instantiation must have a unique labelling.

We have defined four binary geom constructors: union, difference, set difference and geomMap. These are written as

```
g1 + g2
g1 - g2
g1 \ g2
g1 @ g2
```

They all augment the store with a new geom when they succeed. The unary constructor 'flatten' is written

```
#g
```

and creates a geom which has all the leaf-nodes of the original, but no substructure. For example,

```
#<a:<a:A,b:B>, b:(A,B,C)> = <aa:A, ab:B, b1:A, b2:B, b3:C>
```

where the geoms A,B,C are leaves and the underlined labels are formed by concatLabel, that is to say,

```
ab = concatLabel a b
```

5.6.5 Agents

Agents allow us to combine the basic actions. The basic connectives are parallel, sequential and disjunct combinations, written as

a1, a2

a1 ; a2

b -> a2 | a3

where b is a basic action or guard and a1, a2 and a3 are any agents. An agent may be a goal, a composite agent or one of the constraints defined above. For example,

! {upon} (a,b) -> ! g = a | ! g = b

is an agent which first tells that a is upon b and then if it succeeds that g equals a, or if it fails that g equals b.

Goal definitions are written using '::'. Thus, the goal

```
aOrB(a,b,g) ::
        ! {upon} (a,b) -> ! g = a | ! g = b
```

defines an agent which is the same as the previous example. The semantics of goal reduction is equivalent to substitution of parameters, so that for example,

aOrB(g1,g2,g3)

is equivalent to

! {upon} (g1,g2) -> ! g3 = g1 | ! g3 = g2

Finally, we may introduce new undefined geoms into any expression by suffixing their names to the expression, separated from the expression with '^'. So that, for example

```
addOne(g, newg) ::
newGeom ^
        ! newg = g + {newGeom}
```

defines an agent which 'adds' an anonymously labelled, undefined geom to the set g, returning the new value in the argument newg. (Of course, nothing is actually added to g itself, since it must already have a ground label set for the union to succeed.

5.7 Conclusions

In this chapter we have developed a notation for the definition of agents communicating through a constraint store. In the next chapter (Chapter 6) we will illustrate the use of the notation and develop some additional syntactic constructs.

Much of this chapter has been concerned with giving a formal specification of the behaviour of the constraint store in response to the basic operations of ask and tell. This has necessarily been fairly detailed, so we will attempt in this section to take a step back and restate the broader view, in order to show how it relates to the rest of the thesis.

In Chapter 1 we proposed a view of the user interface as a form of visual representation. This view required us to distinguish three different kinds of entity: domain model objects, which are the objects being represented; interface model objects, which represent the domain model; and display objects which are particular instantiations of the interface model. It is this third category which distinguishes dynamic representations, such as graphical user interfaces, from static ones such as diagrams. In the view we are proposing, the state of the interface model is the set of possible displays which correspond to a given domain state (see Figure 5.3).

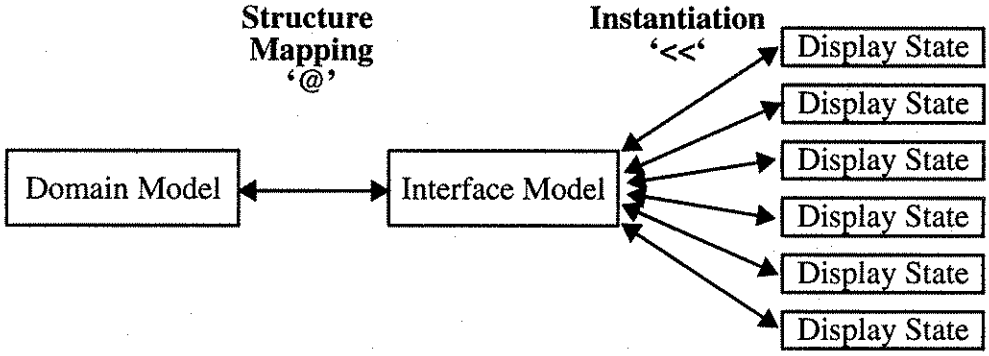


Figure 5.3 The interface as visual representation: there is a one-to-one mapping between interface model states and domain model states and a many-to-one mapping between display states and interface model states.

In order to incorporate this view into a computational framework that can form the basis for specification of interactive systems, we need to show how each of these entities may be supported. In the computational framework we have described in this chapter, there are two basic kinds of constraint: those defined over geom structures (i.e. fixed-width, labelled trees), and those defined over regions in 2.5D space. The spatial constraints provide a convenient means to specify sets of allowed interface states: in fact a constraint is nothing more than a set of allowed states. We can thus use the constraint system to specify the linkage between individual display states and the interface model.

The relationship between display state and interface model is one of

instantiation: we described this in some detail in section 5.4.3. Our formulation of the spatial constraints was such that for any system of regions, exactly one of the topological relations and one of the orientation relations will hold between any two of the regions. Thus, instantiations are characterized by there being a unique relation between any two objects. On the other hand, in the interface model there may be many allowed relations. We have defined the instantiation constraint to represent this relation: e.g. the agent

! displayModel << interfaceModel

asserts that a geom called displayModel is an instantiation of one called interfaceModel.

Similarly, the relationship between interface and domain is one of structural mapping, in which structures in the domain model are mapped to structures in the interface model. We represent this relationship by the geom map constructor e.g.

! link = domainModel @ interfaceModel

asserts that the geoms link, domainModel and interfaceModel have the same labels and that the parts of link are pairings of the parts of domainModel and interfaceModel.

We will explore these relationships in more detail in the next chapter.

Chapter 6 Using the notation

This chapter illustrates the use of the notation which was developed in the previous chapter. In section 6.2 we consider how to express some idioms which we will use frequently. These extend the syntax of the notation, but their semantics is defined in terms of the notation itself. We then consider in section 6.3 how to use the notation to describe some frequently used interaction techniques and properties of the display and windowing systems. Finally in section 6.4 we consider an extended example of the use of the notation, based on an interactive graph-editor application.

6.1 Introduction

In this chapter we develop some examples of the use of the notation which was described in the previous chapter. At the same time we will extend the notation by defining new syntactic structures. However, we will give these a semantics which is defined in terms of the notation itself, so that we do not need to extend the underlying semantics of the notation. Thus, these extensions will not increase the expressiveness of the notation, but will act as shorthand for useful idioms.

We will also illustrate the use of the notation by way of a running example: an interactive editor for graphical representations of directed graphs. In general, there is something of an art to the choice of examples: a good example needs to be simple enough so that the points which it illustrates are not lost in a mass of detail; yet it must also be non-trivial, or else it will not provide enough material to illustrate all the required points. In our case, we specifically wish to show that the notation has applications to the description of each of the components of the arch model (section 2.2.3), so we will develop the example with reference to the principle components of that model.

6.2 Some conventions

In this section, we describe some 'syntactic sugar' that will simplify the definitions given later in the chapter.

6.2.1 Constraint definitions

It will be convenient to be able to define abbreviations for constraint expressions. For example, to say that regions x and y are in contact we would need to say

{overlap, ioverlap, upon, iupon, hide, ihide, coincide, icoincide} (x,y)

We therefore allow the syntactic production

constraintDefinition :: name = = constraintExpression

So that we could instead say

touching(x,y) ==
{overlap, ioverlap, upon, iupon, hide, ihide, coincide, icoincide}(x,y)

The constraints inside the expression are unqualified by '!', '?'. When the definition is invoked as a goal it should be used in conjunction with '!', '?'. The constraints inside the body of the definition are interpreted as either ask or tell accordingly. For example,

! touching(x,y)

is equivalent to

! {overlap, ioverlap, upon, iupon, hide, ihide, coincide, icoincide}(x,y)

and similarly for '?'. The expression may also be compound, so that we might say, for example,

link(x, y, z) == touching(x,y), touching(y,z)

in order to express a connected sequence of objects. In this case,

? link(x,y,z)

would get expanded to

? touching(x,y), ? touching(y,z)

which in turn becomes

? {overlap, ioverlap, upon, iupon, hide, ihide, coincide, icoincide}(x,y),
? {overlap, ioverlap, upon, iupon, hide, ihide, coincide, icoincide}(y,z)

We require that the definitions are non-recursive and contain only parallel conjunctions of basic actions. The semantics of the abbreviation is given by replacing it with its definition.

6.2.2 Geom parts

Two definitions that we shall use frequently test if a geom is a part of a structure. These will mainly be used to check for set membership.

```
includes(structure,geom) ==
        {} = {geom} \ structure
excludes(structure,geom) ==
        structure = structure \ {geom}
```

The first definition checks if the set which contains the single element geom is reduced to the empty set by removing the elements contained in structure. This is just a roundabout way to ask if the element geom is contained in the set. It would seem more natural to write

```
structure =/= structure \ {geom}
```

however, we cannot do this since there is no inequality constraint. The reason for this is that we cannot tell that two geoms are unequal without making an arbitrary choice of labels if either is undefined. Compare this with the case where we tell that an undefined geom is equal to a defined one: in this case the labels of the undefined geom can be determined directly from those of the defined one.

The second definition succeeds if the structure is the same following the set difference with the set containing the single given element. Obviously this will be the case only if the given element is not contained in the set. The constraints have the same effect whether they are implemented as ask or tell. Both constraints suspend until the label set of structure is ground.

We can use set difference to remove parts from a structure. We can also use it to define a constraint which removes those parts which are not in a given set. We will define a constructor '&' to do this.

```
g = g1 & g2
```

The final value of g is g1 with those parts removed which are not in g2. For example

```
<a:A> = <a:A, b:B, c:C> & (A,D,E)
```

Once again this is really just syntactic sugar, since it is equivalent to

```
g = g1 \ (g1 \ g2)
```

This can succeed as a tell when both g1 and g2 are grounded. As an ask constraint it requires all three arguments to be grounded. The new geom g' has the same labelling as g1 for the parts in g2. Other parts of g1 are absent from g2.

Now we can replace one element of a geom with another. The trick is to ensure that the replaced part gets the right label. If we just say

```
geom' = geom \ {part} + {part'}
```

where part has been replaced by part' in the updated geom, geom', then the new part

will have an arbitrary label. So we say instead

```
replace(geom,geom',part,part') ==
labelledPart ^
        ( (geom & {part}) @ labelledPart)._ = (part, part'),
        geom' = geom \ {part} + labelledPart
```

A word of explanation is in order:

```
(geom & {part})
```

is a geom whose one label (call it lab) references part. The map

```
( (geom & {part}) @ labelledPart)
```

is a geom with the label lab which points to an ordered pair whose first member is part and whose second member is the initially undefined element in labelledPart. The latter is a structure with the label lab and undefined element. Finally, we reference the one member of the map with the anonymous label '._' and equate it to the pair (part, part'), thereby equating the element of labelledPart to part'. The second statement removes the old part and replaces it with a new part, now with the correct label.

Sometimes we will want to be able to update a set of parts. We have already seen how to replace an individual element. The following definition takes a geom and a map structure, consisting of pairs of geoms and creates a new geom in which each pair of the map is used to update the geom. The new geom has the same labels as the original.

```
replaceAll(geom, geom', partsMap) ::
old, new, g ^
        ? partsMap = <> -> ! geom = geom' |
        ( ! partsMap._ = old @ new, ! replace(geom, g, old, new),
        replaceAll(g, geom', (partsMap \ (old,new)) ) )
```

In the next section we will give some more sugared syntax for this kind of recursive processing of all elements in a structure.

6.2.3 Distributed constraints

We are using geoms to represent sets of objects as well as individual structured objects. Often we will want to be able to apply a constraint to all members of a set of objects, or to check that a constraint applies to some member of a list. For example, suppose we define an *arrow* as the structure

```
<head,tail,body>
```

and we have a set of arrows represented by a structure named arrows. Now we can

constrain all the arrows to point left to right by the expression

```
leftToRight ( arrows ) ::
a ^
        ? arrows = <> -> stop |
        ! a = arrows._, ! {left}(a.tail,a.head) -> leftToRight ( arrows \ {a})
```

That is to say, in order to distribute the constraint

```
! {left}(tail,head)
```

across the whole group of arrows, we introduce a recursive definition which applies the constraint to one of the group and then recurs on the remaining items in the group. However, it is inconvenient to have to define a separate agent every time we wish to distribute a constraint, so we will introduce the idiom

all e **:** setName **[** agentExpression**]**

which gets translated into the definition

```
% (set) ::
e, set' ^
        ? set = <> -> stop
        | ! e = set._ , set' = set \ {e}, [ agentExpression ]-> % (set ').
```

together with a single invocation of the agent

```
% (setName)
```

Where we are using the percent sign to represent some unique name which is generated for each occurrence of all. Thus, the agent leftToRight would be written as

```
all a : arrows [ ! {left}(a.tail, a.head) ]
```

and gets expanded into the definition

```
agent1234(set) ::
a, set' ^
        ? set = <> -> stop
        | ! a = set._ , set' = set \ {a}, [ ! {left}(a.tail, a.head) ] -> agent1234
        (set').
```

(where agent1234 is the unique name generated for the agent), and a single application of the agent

```
agent1234(arrows)
```

Notice that we execute the agent expression within a guard (square brackets). In this case, it is not strictly necessary to do this because the expression is already a basic action. However, in general we want to be able to use arbitrary agent expressions. The disjunct

[agentExpression] -> % (set ')

requires the agent expression to be a basic action, so we treat it as a guard. The effect of this is that the expression in the disjunction is executed without interleaving. (For a discussion of guards, refer back to section 5.5.2.)

Similarly, we will sometimes want a construction which succeeds when it is possible to apply an agent to any member of a set. For example to check whether there is some arrow which connects two given nodes we could write

```
checkConnect (node1, node2, arrows) ::
a ^
        ? arrows = <> -> stop
        | ! a = arrows._ , ? touching (a.tail,node1) , ? touching (a.head,node2) -
        > stop
        | checkConnect (a)
```

where touching is as defined in the previous section. By analogy with the all construction, we will introduce the syntactic production

any element **:** set [agentExpression]

which gets translated into

```
% (set) ::
e, set' ^
        ? set = <> -> stop
        | ! e = set._ , set' = set \ {e}, [ agentExpression ] -> stop
        | % (set ').
```

So that checkConnect would be written

```
Any a in arrows [ ? touching(a.tail,node1) , ? touching(a.head,node2) ]
```

Notice that in this case, the agent expression contains identifiers that do not refer to the element a. These are bound within the scope of the agent where the distributed constraint is used.

A slightly more complicated example is the following nested expression which asks whether every arrow connects two nodes.

```
all a : arrows [
        any n : nodes [ ? touching(a.tail,n)],
        any n : nodes [ ? touching(a.head,n)]]
```

In the case of all the agent fails as soon as one of the individual agents fails and evaluation of all of the agents is abandoned. In the case of **any** it succeeds as soon as the agent has succeeded with one of the elements and evaluation of the agent with other elements is abandoned.

Suppose now that we wish to collect together the set of arrows which leave a particular node. We could do this by the following procedure definition.

```
outgoingArrows(node, arrows, outgoing) ::
others, a, arrows' ^
        ? arrows = <> -> stop
        | ! a = arrows._, ?touching(a.tail,node) -> ! outgoing = {a} + others
        | ! outgoing = others ),
        arrows' = arrows \{a}, outgoingArrows(node, arrows', others)
```

We will use the following idiom as shorthand for this construction (set comprehension)

collect s : setName **into** c : collectionName [agentExpression]

which is translated to

```
%(setName, collectionName) ::
others, s, c, setName' ^
        ? setName = <> -> stop
        | ( ! s = setName._, [ agentExpression ] -> ! collectionName = {c} +
        others
        | ! collectionName = others ),
        setName' = setName \{s}, %(setName', collectionName)
```

together with an application of the agent

```
%(setName,collectionName).
```

Where c is a placeholder for individual elements of the collection. The term c acquires its value in the agent expression, or it may simply be the same as the first term. For example, the previous example could be written

```
Collect a : arrows into o : outgoing [ ? touching(a.tail,node), ! a = o ]
```

or more simply

```
Collect a : arrows into a : outgoing [ ? touching(a.tail,node) ]
```

All three of all, any and collect succeed trivially when the given sets are null.

The use of the any construction is unstable if we use dont-care commitment in disjunctions (section 5.5.1): consider the following example.

```
a, b, c, r ^
        ! {disjoint} (a, r),
        ! set = {a,b,c},
        any s : set [ ! {upon} (s, r) ]
```

This will fail if the any construction is told before the {disjoint} constraint, because any succeeds by applying the constraint {upon} (a,r) which is inconsistent with

{disjoint} (a,r). On the other hand, if the any construction is applied after the {disjoint} constraint, then it may succeed with {upon} (b,r). The cause of the instability is the disjunction in the definition of any. With don't-care commitment, once the store has committed to telling the constraint {upon} (s, r) to some member of the set, no failure residue is preserved. With don't-know commitment, the system preserves the other branch of the disjunction to be pursued in the case of failure. Thus, in our example, the any construction would commit to

```
!{upon} (a,r)
```

leaving the failure residue (effectively)

```
any s : set \ a [! {upon} (s,r)]
```

which would get pursued once the constraint ! {disjoint} (a, r) had failed. A similar argument applies to collect when it is used with a tell constraint in the agent expression. For example

```
a, b, c, r, set1, set2 ^
        ! {disjoint} (a, r),
        ! set = {a,b,c},
        collect s : set1 into s : set2 [ ! {upon} (s, r) ]
```

once again, this might result in the conjunction failing if {disjoint} (a,r) is told after the collect expression, or succeeding if it is told before.

6.2.4 Geom definitions

The basic categories of object with which we shall define the pictorial aspects of the user interface are regions, structures and constraints. We have already discussed the basic set of constraints with which we shall describe relations between regions. We now consider how they may be combined to form structures and how the behaviour of those structures may be represented.

Geoms are the basic elements of visual representations. A *geom definition* uses a structure to identify named parts. It is likely that the definition of a geom will require more than a set of part labels, probably there will be constraints that must hold between the parts. We can use constraint definitions to define such structures. The following example shows how an arrow may be defined as a geom, including constraints on its various parts.

```
arrow(a) ==
        a = < from, to, head, tail, body >,
        contains(a.tail,a.from), contains(a.head,a.to),
        overlaps(a.head,a.body), overlaps(a.tail,a.body)
```

We can add constraints to an existing definition in order to create a new one. For

example,

```
horizontalArrow(a) ==
        arrow(a) , {hAlign} (a.from,a.to)
```

Additionally, it gives us the possibility of creating geoms which combine properties of others. The union operator '+' allows structures to be combined. The general form of expressions using the operator is

```
Geom = GeomExp1 + GeomExp2
```

which results in a definition equivalent to one with the combined parts of the two expressions and the conjunction of their constraints. The following example involves defining standard sets of landmark points to serve as 'handles' for direct manipulation and layout. Thus,

```
handles(h) ==
        h = < frame,n,s,e,w,ne,nw,se,sw >,
        {weakAbove}(h.n,h.frame),{weakAbove}(h.nw,h.frame),
        {weakAbove}(h.ne,h.frame),
        {weakBelow}(h.sw,h.frame), {weakBelow} (h.s,h.frame),
        {weakBelow}(h.se,h.frame),
        {hAlign} (w,e), {weakRight}(h.e,h.frame),
        {weakRight}(h.ne,h.frame), {weakRight}(h.se,h.frame),
        {weakLeft}(h.w,h.frame),{weakLeft}(h.nw,h.frame),
        {weakLeft}(h.sw,h.frame), {vAlign} (n,s)
```

Now we can use this definition in composition with others, e.g. given that we have

```
textBox(t) ==
        t = < box, text, outline>,
        {upon}(t.text,t.box),
        {justUpon}(t.outline,t.box).
```

we can use the operator '+' to combine the textBox and handles, as follows.

```
textForLayout(t) ==
b,h ^
        textBox(b), handles(h),
        t = b + h,
        {coincide}(t.box,t.frame)
```

Common elements of descriptions may be factored out and merged with other components. This simplifies definitions and facilitates the creation of generic behaviours: a TextForLayout geom may form part of structures in the role of a TextBox or Handles. The definition is effectively a constant structure, so that, for example,

```
! textForLayout(t)
```

succeeds by creating the TextForLayout structure then applying the constraints in the definitions and finally equating the resulting structure with t. That is to say, it is equivalent to the following constraint application

```
! {weakAbove}(t.n,t.frame),! {weakAbove}(t.nw,t.frame),
        ! {weakAbove}(t.ne,t.frame),
        ! {weakBelow}(t.sw,t.frame), ! {weakBelow} (t.s,t.frame),
        ! {weakBelow}(t.se,t.frame),
        ! {hAlign} (w,e), ! {weakRight}(t.e,t.frame),
        ! {weakRight}(t.ne,t.frame), ! {weakRight}(t.se,t.frame),
        ! {weakLeft}(t.w,t.frame),! {weakLeft}(t.nw,t.frame),
        ! {weakLeft}(t.sw,t.frame), ! {vAlign} (n,s)
```

On the other hand,

```
? textForLayout(t)
```

succeeds if the structure created by the definition may be equated with t. That is to say, the ask is equivalent to the following constraint application

```
? {weakAbove}(t.n,t.frame), ? {weakAbove}(t.nw,t.frame),
        ? {weakAbove}(t.ne,t.frame),
        ...
        ? {weakLeft}(t.sw,t.frame), ? {vAlign} (n,s)
```

Where the constraints from the definition are exactly as before, but instead of telling the constraints on t, this time we ask them.

6.2.5 Sets objects

The basic operations which we allow on sets are union and difference. We discussed these in the previous chapter. We may check for set membership by the use of the constraint expressions defined in section 6.2.2.

```
? includes (set, member)
? excludes (set, member)
```

The first expression succeeds if the member is an element of the set and the second succeeds if it is not.

We will use sets to model the objects of interest in the interface. A transition between interface states may be the result of a change to one of the sets of objects.

In the context of modelling user interfaces, we will often need to model sets of objects which share common properties, so that it is a condition of membership of the set that the element have the given property. In this case, provided we can tell the given property onto a new element, that element may be added to the set.

For example, if we wanted to define a set of arrows, we could do so by

defining a structure constant Arrow as before, together with the global variable Arrows. Then we would write

```
updateArrows(additions, deletions) ::
        all element : additions [ ! arrow(element) ] ,
        ! Arrows' = Arrows + additions \ deletions
```

We may also wish to describe sets of objects which are defined intensionally: that is to say, the set membership is derived from some other set or sets by including those members which satisfy a given property. For example we might wish to model the set of arrows which originate in any of a set of boxes. Thus, suppose we are given two sets, boxes and arrows, then we could define the set outgoingArrows as follows

```
collect a:arrows into a:outgoingArrows [
        any b:boxes [? touching (a.from, b) ] ]
```

The difference between intensional and extensional sets is subtle, but important. Both forms of definition result in sets whose members share a common property: the property is *necessary* for set membership in both cases. However, in the case of intensional sets, it is also *sufficent* to define set membership: any group of objects satisfying the property will be a member of the set. When sets are defined extensionally, possession of the membership property does not guarantee membership of the set.

6.2.6 Disjunction

The general form of disjunction is

```
b -> a1 | a2
```

where b is a basic action and a1 and a2 are arbitrary agents. If b succeeds then a1 is pursued, if b fails then a2 is pursued.

The expression

```
b1 -> stop | b2
```

succeeds if either b1 or b2 succeeds. Similarly, to make a choice between n basic actions, we could write

```
b1  -> stop | b2 -> stop | b3 -> stop | ... | bn
```

The b1 may be arbitrary guards or they may be ask or tell constraints. Thus to choose between n agents we would write

```
[a1]  -> stop | [a2] -> stop | [a3] -> stop | ... | [an]
```

where the agents are guarded (shown by square brackets). We will adopt the following shorthand for this expression

a1 : a2 : a3 : ... : an

That is to say, the ai are executed in guards within a sequence of disjunctions.

6.3 Generic interactors

In this section we show how the constraint-based approach described in previous chapters may be used to describe some commonly used interaction components and techniques. In terms of the reference model described in chapter 2, these are presentation objects.

The agents which we will describe are domain-independent: they require no explicit knowledge of domain properties and behaviour. Communication with domain objects is mediated through the constraint store. At a later stage, we will have to ask how these agents come into being and how they are assigned the structures which are their parameters. These operations are the role of the dialogue-control objects of the Arch Model.

In this section we consider how to model the display state using agents. The display at any time is an instantiation of part of the store. In the next section we will show how to structure user interface specifications by reference to the Arch Model. In that section we will model the presentation component as a mapping from interface objects to specific instantiations.

We will show later how windowing systems and visibility may be represented, for the moment we will assume that the whole of the instantiation is visible. There will be a set of objects whose permissible states are to be represented in the store. Our model therefore has a sequence of instantiations and a set of displayed objects. The current state of the display is an instantiation of the set of objects in the display. The interface model, of which the display is an instantiation, may be structured in a way that is specific to the particular interface. However, the display itself is an unstructured mapping between regions in the interface model and the corresponding instantiations. This is examined in more detail in section 6.4.4, when we consider an extended example.

6.3.1 Object selection

Object selection in typical direct manipulation interfaces is of one of three kinds: selection of a single object; sequential selection of a group of objects (e.g. by shift-click); simultaneous selection of a group of objects (e.g. by dragging out an area). We will show how each of these idioms may be represented using the notation developed in the previous chapter. We are not concerned with the low-

level details of input, such as which keypresses activate selection.

We first consider the simple case where it is required to pick a single object. We need to be able to identify the uppermost object from among a group of objects. The following two constraint definitions allow us to do this. First of all, visible, which succeeds if a distinguished region is not part covered by any other object in a given set of geoms.

```
visible(r, set) ::
all s : (#set \ {r}) [ ? {upon, hide, overlap, disjoint } (r,s) ]
```

Notice that the agent cannot be used to bind a value to r, since the distributed expression suspends until r and set are known. We flatten the given set of geoms, since they may not all be region valued.

We now use the definition of visible in a basic action definition which picks out one of the visible regions in a set of geoms. It is possible that several objects in the set may satisfy the definition of visible, for example when the elements of the set are disjoint regions. In this case, the picked object will be chosen arbitrarily.

```
pickVisible(r, set) ::
any s : #set [ visible(s, set) , ! s = r ].
```

Notice that this definition may be used to bind a value to r, since we tell the value explicitly. In the case where the given set is null, pickVisible leaves the selection object undefined.

Finally we define an agent which selects the uppermost region at a point.

```
pick(point,part,set) ::
picked ^
        collect r : #set into r : picked [ ?{upon} (point,r) ] ,
        pickVisible(part, picked).
```

In this example, picked is the set of regions which lie under the selection point. The agent forms this set and then selects a visible element of the set (which will be the uppermost one). The agent

```
pickVisible(part, picked)
```

suspends until the subset of elements has been determined. It then binds a value to part, which will be one of the elemental regions of the set of objects. The agent pick leaves part undefined if there is no object at the selection point.

Note that once we have picked a region, it is straightforward to identify the geom from which it came. e.g.

```
pick(p,r,objects), any o:objects [ ? includes(o,r) ! geom = r ]
```

finds the geom which contains the picked object.

We now consider multiple selection: first of all sequential selection. The most natural approach to defining this form of selection is as a series of simple selections. However, the way in which structures have been defined requires them to have known size. Of course, we cannot determine in advance how many objects will be selected. We get around this problem by having an agent which accumulates a set of selected objects, recurring each time a new object is added to the set. The following definition half-way solves the problem, but has a couple of flaws.

```
seqPick(point, set, currentObjects, selection) ::
object, finalSelection ^
        pick(point, object, set),
        seqPick(point, set, currentObjects + {object}, selection)
```

The implicit ask constraint in the structural expression

```
currentObjects + {object}
```

ensures that the recurrence of the agent suspends until an object has been selected. Meanwhile the agent seqPick continues to recur on itself accumulating more and more objects in the set of current objects. We need some means to prevent the recurrence of the agent by signalling that the selection process has terminated. We will achieve this by nulling the selection point, thus.

```
seqPick(point, set, currentObjects, selection) ::
object ^
? point = <> ->
        ! selection = currentObjects |
        pick(point, object, set),
        seqPick(point, set, currentObjects + {object}, selection).
```

This solves the problem of termination, because when the selection point is null the agent stops recurring and a structure value for the term selection finally gets passed back to the original agent. However, there is still one other major flaw: the selection point never changes. We therefore need the agent to recur on a new copy of the selection point, so we will require a way to model persistence of objects. We have referred to one way to do this in section 3.3.3.

```
seqPick ( point, set, currentObjects, selection) ::
object, p, point' ^
```

```
? point = <> ->
        ! selection = currentObjects |
        ! point = (p, point'),
        pick(p, object, set),
        seqPick(point', set, currentObjects + {object}, selection) .
```

We have modelled point as a tree structure. Each value of point has two parts: a region-valued term and a term with, recursively the same structure. This approach to modelling persistent object state within a constraint-based language was referred to in section 3.3.3. Instead of passing a single node as parameter to the agent seqSelect we pass a sequence of values. It is presumed that this will eventually get bound to another structure of identical form, or ultimately to the null structure.

Finally, we show how to define the selection of multiple regions in a single action. Here the selected regions are those which overlap with a given region. We could just define this in the same way as pick, using a single region (rather than a sequence of regions).

```
multiPick (region, set, selection ) ::
collect r:#set into r:selection [ ? {overlap, hide, upon} (region,r) ]
```

However, we may want to allow the selection region to change, while some action (such as highlighting occurs). So we will define the selection procedure using a series of regions, allowing the agent to recur when the selected set has been ground. We achieve this by guarding the distributed constraint. We will use the same method as before to signal termination of the selection procedure.

```
multiPick (region, set, selection) ::
r, s, region', selection' ^
        ! region = (r, region'),
        ! selection = (s, selection'),
        collect r:#set into r:selection [ ? {overlap, hide, upon} (region,r) ],
        ? region' = <> -> stop | multiSelection (region', set, selection').
```

In this example, the selection is represented as a sequence of structures.

6.3.2 Movement

We can represent movement of objects in two distinct ways: relative movement, in which we are interested in the displacement of an object relative to its initial position; and absolute movement, in which we specify a new position for the object.

Relative movement may be specified within the qualitative constraint framework by the use of orientation constraints. So, for example, we may define

```
moveRight(region, region') ::
        ! {weakLeft } (region, region')
```

This allows us to define relative movement for single regions. However, we will more usually be dealing with geoms rather than individual regions, so we will need to define movement for composite objects. With composites there are many more possibilities for movement, because of the possibility of relative motion of the components. We will define an agent to achieve rigid translation of composites, but we need to be explicit about preserving the relative positioning of every element of the composition, as well as defining the translation over the elements of the composite. To achieve this we use the map ('@') constraint, together with the instantiation constraint '<<'.

```
moveRight(geom, geom') ::
        ! geom << geom' ,
        all (r , r'): ( #geom @ #geom' ) [! {weakLeft } (r,r')]
```

The first constraint says that both the structure of the geom and the spatial constraints between the parts of the geom are preserved. The map constraint requires that one or other of the geoms be ground. In fact if the first geom is an instantiation, this condition is satisfied and the constraint implies that the second is also an instantiation. The second constraint says that any region-valued part in the initial state of the geom is to the left of its continuation.

We may define movement in other directions similarly.

```
moveLeft(geom, geom') ::
        ! geom << geom',
        all (r , r'): (#geom @ #geom') [! {weakRight} (r,r')]
moveUp(geom, geom') ::
        ! geom << geom',
        all (r , r'): (#geom @ #geom') [! {weakBelow} (r,r')]
moveDown(geom, geom') ::
        ! geom << geom',
        all (r , r'): (#geom @ #geom') [! {weakAbove} (r,r')]
```

We can define horizontal and vertical movement as

```
horizontalMove(g, g') :: moveLeft(g, g') : moveRight(g, g')
verticalMove(g, g') :: moveDown(g, g') : moveUp(g, g')
```

Finally, we define movement as a parallel conjunction of horizontal and vertical movement, either jointly or separately.

```
move(g, g') :: (verticalMove(g,g'), horizontalMove(g,g'))
            : verticalMove(g,g')
            : horizontalMove(g,g')
```

A useful way to describe movement is to have one object track another, typically a point, as occurs when dragging. This is another example of relative movement, because the relation of the tracking objects to the tracked object remains the same. We can express this using move. We define a composite object which consists of the tracked object and the tracking object. We assume that the tracking and tracked objects are tree-structured, as with the selection examples.

```
track(geom,trackPoint) ::
t1,t2, trackPoint',g1,g2,geom' ^
            ! trackPoint = (t1, (t2,trackPoint')),
            ! geom = (g1,(g2,geom')),
            move(g1 + t1, g2 + t2) : ! (g1 + t1) = (g2 + t2),
            (trackPoint' = <> -> stop ! track((g2,geom'),(t2,trackPoint'))
```

6.3.3 Least surprise

An important issue in the description of dynamic change is to define the effects of the change on those objects which are not moved directly. Consider the following simple example.

{upon, overlap, disjoint} (b,a)

{upon} (c,a)

The position of b relative to a is less constrained compared with that of c. Now suppose that, given the initial layout shown in Figure 6.1, region a is dragged. The question is what should happen to the regions b and c ? Both of the final states depicted in the figure (i and ii) are valid instantiations of the given constraints.

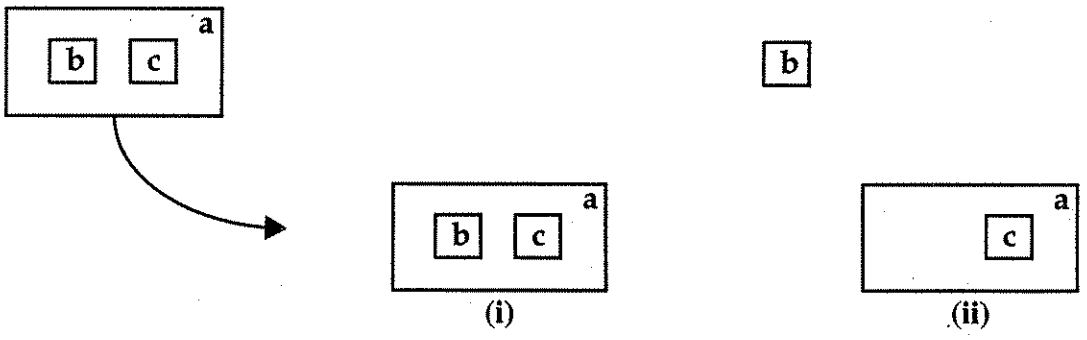


Figure 6.1 Two possible outcomes of dragging region **a.** Constraint-based specifications require some means to choose between possible solutions.

We also need to consider the allowed intermediate states of the objects while

being dragged. Suppose that the constraint between regions b and a were

{upon, disjoint} (b,a)

Now, in order to move *continuously* from the initial position to position (ii), it is necessary for there to be an intermediate state in which the constraint

{overlap} (b,a)

is allowed. Thus, in this case, the state shown in (ii) is unreachable, even though it is a feasible instantiation.

These simple examples illustrate some of the issues involved in characterizing the dynamic behaviour of interface objects. We will now look at how such behaviours may be represented in the notation which we have developed.

Loosely speaking, what we require is that as much as possible of the layout should remain constant. This may be referred to as 'the principle of least surprise'. It is a characteristic of constraint-based descriptions that there may be multiple solutions to a given problem. Therefore it may be necessary to specify which of the possible solutions is to be preferred.

Unfortunately, the term 'least surprise' is loaded. It begs a number of questions of a psychological nature. Principally, it assumes that we know what the user expects to happen: that is it assumes that we know something about the user's model of the system. As we have observed in the introduction to the thesis, we will not pursue such psychological issues; instead our approach is to provide sufficient expressiveness in the notation to allow the user interface designer to define a wide range of user interfaces, presuming that this will include the ergonomically 'good' ones as well as those that are unsatisfactory from the point of view of usability. We do not, of course, intend this omission to suggest that the psychological issues are unimportant or irrelevant to interface design.

We will therefore consider least surprise from a different point of view: that of the similarity of two instantiations and we will consider ways in which such similarity may be measured.

In the case of the simple example, we would expect that in dragging a file icon the relative configuration of other objects in the display would not change. In this case, we actually require a stronger condition, namely that the regions occupied by those objects should not change as a result of dragging. For example, in order to say that one object (obj) moves and that the other objects in the display remain unchanged, we could use the construction

! move(obj, obj') , ! replace(objects,objects',obj,obj')

This replaces the moved object by the new version, keeping the other objects the same.

We can use this construction to define an agent which acts upon the handles defined in section 6.2.4, so as to resize it. We assume that the given object is called obj and has some substructure equal to handles. For example,

```
rightHandResize(obj, obj') ::
h, h' ^
        ! handles(h'), ! handles(h),
        ! obj << obj', ! obj = h + obj, ! h' = obj' + h',
        horizontalMove(h.e, h'.e), ! h.nw = h'.nw, ! h.sw = h'.sw
```

The instantiation constraint

! obj << obj'

says that the new version of the object has the same structure as the old and that the relative configuration of its parts is unchanged. The statements

! obj = h + obj, ! h' = obj' + h',

bind h and h' to substructure in obj and obj' . The two statements

! h.nw = h'.nw, ! h.sw = h'.sw

constrain the left hand side of the object to be unmoved. We can define similar agents to resize the top, bottom and left sides.

6.3.4 Drag and drop

In drag-and-drop, operations on the current instantiation (presentation layer) may result in changes in the interface model. These may in turn be propagated to the domain model. For example, in a typical file manager application, when a file icon is dragged over a directory icon and dropped onto it, this is interpreted as a command to copy the file into the directory.

While dragging the file icon, feedback is provided when the file passes over a directory to which the corresponding file can be copied. We therefore need to allow for the possibility that the instantaneous state of the display is shared between several agents. On the other hand, the state of the display does not necessarily represent a committed state of the store: it may be only one of several possible instantiations.

When the dragged object is finally dropped, some action is required. There are alternative ways in which we could achieve this. For example, we could arrange for some named interface action to occur in response to dropping the dragged

object. In fact, we could even communicate directly with the domain model. Such an approach is at least direct and easily understood, but suffers the serious drawback that the definition of the interaction is application specific and therefore not readily reusable. Separate definitions of drag and drop would be required for each different user action. If our aim is to provide truly generic interactors, then the definitions of user interactions must be free of domain-specific actions. Generic interactors are desirable insofaras they offer separation (see section 2.2.1). We will show in this section how we can define generic versions of drag and drop in which the action performed is to apply a constraint to the interface model.

The way in which we shall represent drag and drop is as an exploration of the state space of the store: while dragging, the instantaneous state of the display represents a possible instantiation of the constraints in the store. The action of dropping results in the application of a constraint to the objects being dragged and the objects lying at the point where they are dropped. Changes in the instantaneous state of the display during dragging (feedback) have no effect on the domain objects of the application, although they convey information to the user about the possibility of change.

We shall assume for the moment that a single object is to be dragged: subsequently we shall show how to extend the definition of dragging to include multiple objects. Typically, the region selected for dragging will be part of a structure; similarly the target object selected by the action of dropping the dragee will normally be part of a larger structure. We will assume that the constraint which is finally applied is {upon}, although obviously we could define variations which applied a different constraint as the final action.

We have already seen how to represent movement of objects. In defining movement, we are concerned with the relations between successive display states, each of which is an instantiation. With drag and drop, the situation is different because we also need to consider the interface object which is instantiated in the display. Thus, whereas movement is defined by applying constraints between successive display states, without reference to the interface model (see Figure 5.3), the effect of dropping an object is to apply a constraint between objects in the interface model.

The parameters for the operation of dragging are

- a stream of points;
- the dragged object in the interface model;
- the other objects in the interface model;
- a tree-structured stream of display states, each of which instantiates the interface model.

The operation of dragging is

```
drag((p1,(p2,point')),geom, im, (d1,(d2,display')) ) ::
r, r' ^
        ! d1 << im , ! d2 << im,
        any (g,i) : (#im @ #d1) [ ! g = geom ! i = r ],
        any (g,i) : (#im @ #d2) [ ! g = geom ! i = r' ],
        ! move((p1,r) , (p2,r'))
```

The first two constraints state that successive display states are instantiations of the interface model. The next two say that the instantiation of the leaf geom in the current display state is r and in the next display state it is r′ . The final constraint says that r moves to position r', tracking the point. The action of dropping is slightly simpler as we do not have to represent successive states of the display.

```
drop(point, geom, im, display ) ::
r , target ^
        any (g,i) : (#im @ #display) [ ! g = geom ! i = r ],
        pick(point, target, (display \ {r}) ),
        ! {upon}(g,target)
```

As before, r is the instantiation of geom. To drop an object, we constrain it to be upon the top region at the given point. Notice that although we pick out the target object by reference to the current display state, the constraint {upon} is applied to the *interface model* for the dragged objects and target.

Finally, we can put the two agents together, as follows.

```
dragAndDrop( (p, point'), g, im, (d,display') ) ::
g' ^
        ! g = <> -> pick(p,g',im), dragAndDrop((p, point'), g', im, (d,display'))
        | ? point = <> -> drop(p,g,im,d)
        | drag( (p, point'), g, im, (d,display') ), dragAndDrop(point', g, im,dis-
        play')
```

which says that if the dragged object g is undefined (we check this by equating it to the null geom), we pick an object g' and the agent recurs on it. If there are no more points in the stream of point values, we drop the object, otherwise we drag it.

6.3.5 Windowing systems and visibility

Windowing systems increase the complexity of the interface, because we have to take account of which parts of the interface are visible at any time. In fact, up till now, we have assumed that the whole of the instantiation is accessible to the user. If we are to use the notation to reason about the user's interaction with the system, then we need to know in the context of any action which parts of the current instantiation are visible.

The basic model of a window is quite simple: the window consists of a visible area and a set of displayed regions (the object list). A windowing system is a collection of windows. We require that the objects in a given window are not interleaved with those in any other window. We achieve this by imposing the constraint that every object is over (resp. under) another window if its own window is over (resp. under) that window.

```
window(w) ==
        w = <objectList, displayArea> .

windowSystem(windows) ::
        all w1:windows [
            all w2:windows [
            ! window(w1),
            all o:w1.objectList [
                (!{over}(o,w2), !{over}(w1,w2)) :
                (!{under}(o,w2), !{under}(w1,w2))]]
```

A region is in view if it overlaps some part of the display area. (Notice that this does not necessarily mean that the region is visible to the user: to determine visibility, we must consider whether it is hidden by other geoms in the system.)

```
inView(w, r) == {overlap,upon, justUpon, cover}(r,w.displayArea)
```

An object which is in view is visible if it is not obscured by any other object in the same window, nor by any other window. The system of spatial relations which we are using does not permit us to define regions constructively, so we will say that an object is visible if it is not part-covered by any other object. Objects which are part-covered may be partly visible, or they may be completely covered by a combination of other objects. We have no way to distinguish these cases, since we cannot form the union of those objects which part-cover a given region. So we will treat region-valued objects as *visible* whenever they are not part-covered by another object.

```
visibleRegion(ws,win,r) ::
displays ^
        collect w:ws into d:displays [ ! d = w.displayArea ],
        others = #win.objectList + #ws
        ! includes(ws,win),
        ! includes(#ws.objectList,r),
        ! inView(r),
        all o:(#ws.objectList + #displays )[ inView(o) -> ! {disjoint, overlap,
        cover, upon, justUpon}(r,o) | stop ]

partVisible(ws,win,g) ::
        any r: #g [ visibleRegion(ws,win,r) ]
```

In this definition, a geom is *part visible* if some of its parts are visible. A geom is *obscured* if none of its parts is visible.

```
obscured(ws,win,g) ::
        any r: #g [ visibleRegion(ws,win,r) ] -> fail | stop
```

Note that an object may be obscured, yet have parts which the user can see. We can adjust the extent to which an obscured object is visible to the user by varying the number and location of its parts.

6.4 A graph editor

We now develop a simple graph editor using the notation and then show how the basic editor can be refined by additional constraints to achieve different kinds of layout. Finally we give the editor a semantics by linking it to a set of external actions for updating a file system. The aim of this example is to illustrate how the notation allows us to separate out application semantics (the file system), dialogue structure (the basic operations of the editor) and presentation specific concerns (layout detail). We will follow the arch-model framework which was described in the review of architectures for UIMS in Chapter 2.

6.4.1 The domain model

The domain model for our graph editor consists of a set of nodes and a set of directed arcs linking those nodes. The set of nodes is a basic set: we may have unconnected nodes and we are free to add nodes or to remove them without restriction. On the other hand, the set of arcs is a dependent set, since it makes no sense to have an arc which does not connect two nodes. Thus we cannot create an arc independently of the nodes it connects, and removing a node must result in the removal of the arcs connected to it.

Note that when we use the term 'arc' we are referring to an abstract relation

between objects in the domain model: we are *not* referring to any graphical entity which may be used to represent the existence of an arc between two nodes. Subsequently, when we come to consider the presentation component of the graph editor, we will need to consider how arcs may be represented, and then we will use a geom structure called Arrow. We will permit arrows to exist independently of nodes.

The basic operations which may be performed are as follows:

- to create a node;
- to remove a node, in which case the set of arcs must be updated accordingly;
- to create an arc between two given nodes, which requires that the nodes exist already;
- to remove an arc between two given nodes;
- to merge two nodes, so that all incoming (resp. outgoing) arcs to the originals are incoming (resp. outgoing) to the merged node;
- to transfer one node to another, so that all incoming arcs to the first node are deleted, all outgoing arcs remain, and there is an arc incoming to the first node from the second.

So we start with the definitions for node creation and removal. We define the domain model as a set of nodes and a set of arcs. An arc is defined in terms of a pair of nodes, thus

```
arc(a) == a = <from,to>
graphEditorDM(dm) ::
        ! dm = <nodes,arcs>,
        all a : dm.arcs [ ! arc(a), ! dm.nodes = {a.from, a.to} + dm.nodes) ]
```

This says that an arc is a labelled pair from one node to another. The labelled parts of the arc must exist in the set of nodes. We define an expression to create a node, given two domain models (the before and after states).

```
createNode(dm, dm', n ) ::
        ! dm'.nodes = dm.nodes + {n},
        ! dm. nodes = dm.nodes \ {n}
```

which says that the set dm'.nodes contains n, but the set dm.nodes does not. That is, the set has been augmented with n. The expression

```
dm.nodes = dm.nodes \ {n}
```

only succeeds if n is not a member of dm.nodes. This constraint ensures that n is not a member of dm.nodes before the operation. Removal of nodes is achieved in a sim-

ilar fashion. In this case, we need to maintain the arcs consistent when a node is deleted. The constraints in the definition of the graphEditorModel ensure that every arc must connect two nodes in the set dm'.nodes. Thus, the arcs connected to the deleted node will no longer exist in the new model. However, we also need to ensure that no arc is deleted or added to other pairs of nodes. We therefore need to be explicit about the effects of the removal of a node on the set of arcs, as follows

```
deleteNode(dm, dm', n ) ::
del1, del2 ^
        ! dm'.nodes = dm.nodes \ {n} ,
        ! dm. nodes = dm.nodes + {n},
        collect a:dm.arcs into a:del1 [ ? a.from = n ],
        collect a:dm.arcs into a:del2 [ ? a.to = n ],
        ! dm'.arcs = dm.arcs \ (del1 + del2)
```

When adding an arc, we need to specify the from and to nodes. These are passed as a single parameter to the expression, with the same structure as defined for arcs.

```
createArc(dm, dm', a) ::
        ! dm.arcs = dm.arcs \ {a},
        ! dm'.arcs = dm.arcs + {a}
deleteArc(dm, dm', a) ::
        ! dm'.arcs = dm.arcs \ {a},
        ! dm.arcs = dm.arcs + {a}
```

One node is merged into another by swinging all incoming arcs pointing to the first node to point to the second and all outgoing arcs from the first to be outgoing from the second. The first node is deleted following the change.

```
mergeNode(dm, dm', n1, n2) ::
in, out, in', out' ^
        ! dm'.nodes = dm.nodes \ {n1},
        ! dm'.arcs = dm.arcs \ (in + out) + (in' + out'),
        collect a:dm.arcs into a:in [ ? a.to = n1] ,
        collect a:dm.arcs into a:out [ ? a.from = n1 ],
        collect a:in into i:in' [ ! (a.from = i.from, i.to = n2) ],
        collect a:out into o:out' [ ! (a.to = o.to, o.from = n2) ]
```

The operation moveNode is similar, except that only the outgoing arcs are transferred, while the incoming arcs are removed. A new arc is created from the target node to the moved node.

```
moveNode(dm, dm',n1, n2 ) ::
```

```
in, out, in', out' ^
        ! dm'.nodes = dm.nodes \ {n1},
        ! dm'.arcs = dm.arcs \ (in + out) + (in' + out'),
        collect a:dm.arcs into a:in [ ? a.to = n1 ] ,
        collect a:dm.arcs into a:out [ ? a.from = n1 ],
        collect a:in into i:in' [ ! (a.from = i.from, i.to = n2) ],
        collect a:out into o:out' [ ! (a.to = o.to, o.from = n2) ]
```

The abstract behaviour of the graph editor may be expressed by the following program, which says simply that any change to the domain model of the graph editor is one of the operations which we have just defined.

```
graphEditorDomain(dm, dm')::
n, n1, n2, a ^
        (graphEditorDM (dm), graphEditorDM (dm')),
        ( createArc (dm, dm', a) : deleteArc (dm, dm', a)
        : mergeNodes (dm, dm', n1, n2) : moveNodes (dm,dm',n1,n2)
        : createNode (dm, dm',n) : deleteNode (dm, dm',n) )
```

It is possible to add additional constraints to the behaviour of the editor, without having to redefine the basic behaviour. For example, in order to say that following the addition of a node, an arc must be created which connects to that node, we could write

```
graphEditor1(dm, dm') ::
dm1, a, n ^
        ( graphEditor(dm,dm1), createNode(dm,dm1,n) ) ;
        ( graphEditor(dm1, dm'), createArc(dm1,dm',a),
        ! (a.from = n : a.to = n) )
```

The use of agents based on constraints enables us to use the same notation to define the state behaviour of the domain model and its process behaviour.

6.4.2 The interface state

So far, we have described changes which occur in the domain model of the application. This much could apply to any system, whether interactive or not. Of course what we wish to show is that the constraint-based approach allows the definition of the relationship between domain model and interface state. In this section we show how the interface state may be described and linked with the domain model.

The interface we shall describe is one in which boxes and arrows may be linked together by constraints on their position. Loosely, boxes represent nodes in the domain model and arrows represent arcs. However, the mapping between

arrows and arcs is not simple, since we shall allow arrows to exist on their own in the interface state, representing partially completed arcs. The domain model has no such entity as a partially completed arc. Thus, it is not arrows which represent arcs, rather it is a structure comprised of two boxes and an arrow, constrained to be in contact, which represents an arc.

Operations on the interface model, as with the domain model, consist of addition and deletion of objects (boxes and arrows in this case). However, the definition may be looser, since we do not need to require the same consistency constraints in the interface model. For example, to delete a node does not require that the associated arrows be removed. Effectively, the interface model is a specialized drawing package. We might wish to include additional behaviours and we will see later how this can be done. First we consider the basic behaviour of the interface model.

It is worth repeating at this point that what we are modelling is the set of possible display states: we will consider in another section how individual display states may be modelled.

```
arrow(a) == a = < to, from, head, tail, body >

box(b) == region(r)

graphEditorIM(im) :: ! im = <boxes, arrows>

% touch(a,b) == {upon,overlap}(a,b)

% link(b1, b2, a) ==
        arrow(a), box(b1), box(b2),
        touch(a.from, r1), touch(a.to, r2)
```

The next four agents add and delete boxes and arrows.

```
createBox(im, im', b) ::
        ! box(b),
        ! im.boxes = im.boxes \ {b},
        ! im.boxes' = im.boxes + {b}

deleteBox(im, im', b) ::
        ! im.boxes = im.boxes + {b},
        ! im.boxes' = im.boxes \ {b}

createArrow(im, im', a) ::
        ! arrow(a),
        ! im.arrows = im.arrows \ {a},
        ! im.arrows' = im.arrows + {a}
```

```
deleteArrow(im, im', a) ::
        ! im.arrows = im.arrows + {a},
        ! im.arrows' = im.arrows \ {a}
```

We also need to 'unconstrain' objects. For example, in order to detach one end of an arrow and swing it to point to another node, we first need to remove the constraint on the arrow that it touch the first node. We have already discussed the monotonic nature of the constraint paradigm which we are using: once constraints have been asserted, they cannot be removed. This contrasts with other uses of constraints in UIMS work, where there are specific mechanisms for the retraction of constraints. If constraints are to be retracted, it adds considerable overheads to the problem of maintaining consistency in the constraint network. The problem is similar to those encountered in truth maintenance systems based on default reasoning. Hernandez and Zimmermann describe the similarities between default logics and spatial representations based on qualitative constraints [72]. The use of monotonic constraints is simple to implement and gives rise to a simple operational semantics. However, in order to 'remove' a constraint in this paradigm, it is necessary for an agent to recur with a fresh copy of one of its parameters.

Thus, to remove the constraints on the end of an arrow we can say

```
freeHead(im, im', a, a') ::
        ! im.arrows = im.arrows + {a},
        ! im.arrows = im.arrows \ {a'},
        ! im'.arrows = im.arrows \ {a} + {a'},
        ! a.from = a'.from
freeTail(im, im', a, a') ::
        ! im.arrows = im.arrows + {a},
        ! im.arrows = im.arrows \ {a'},
        ! im'.arrows = im.arrows \ {a} + {a'},
        ! a.to = a'.to
```

Note that we have to state explicitly which parts of the arrow retain the original constraints.

We can add constraints without the necessity of copying the object, so that for example, we can constrain head or tail of an arrow to touch a node, as one of the steps in an interaction which creates a structure consisting of two boxes linked by an arrow, representing an arc. So,

```
fixHead(im, a, n) ::
        ! im.arrows = im.arrows + {a},
        ! im.nodes = im.nodes + {a'},
        ! touch ( a.to, n )
```

```
fixTail(im, a, n) ::
        ! im.arrows = im.arrows + {a},
        ! im.nodes = im.nodes + {a'},
        ! touch ( a.from, n )
```

Notice that in these two cases, since we are not altering structure, we do not need the agent to recur on a new version of the interface model. Up till now, all the examples we have seen for the domain model and the interface model have been concerned with changing structure by adding or removing elements from sets. Since we have adopted a very simple model of structures which cannot be modified incrementally, it has been necessary to use this technique for expressing change. Similarly when we wished to remove a constraint, it was necessary to copy the interface model. In this case, since we are adding constraints incrementally, we do not need to copy the model.

We may also constrain one box to lie upon another. We will define a separate name for this action, for convenience.

```
dropBox(im,b1,b2) ::
        ! im.boxes = im.boxes + {b1},
        ! im.boxes = im.boxes + {b2},
        ! {upon} (b1,b2)
```

In fact, this has the effect of transferring all arrows from one node to another (but not vice versa). This can be shown by expanding the definition of link. Thus, what we wish to prove is that

```
! link(b1,b2,a), ! dropBox(im,b2,b3)      ! link(b1,b3,a)
```

By expanding link and dropBox, the LHS of the expression becomes

```
! arrow(a), ! box(b1), ! box(b2),
        ! touch(a.from, b1), ! touch(a.to, b2),
        ! im.boxes = im.boxes + {b2},
        ! im.boxes = im.boxes + {b3},
        ! {upon} (b2,b3)
```

and by further expanding touch we get

```
! arrow(a), ! box(b1), ! box(b2),
        ! {upon, overlap} (a.from,b1), ! {upon, overlap} (a.to, b2),
        ! im.boxes = im.boxes + {b2},
        ! im.boxes = im.boxes + {b3},
        ! {upon} (b2,b3)
```

From which, we can derive

```
! {upon, overlap} (a.from,b1), ! {upon, overlap} (a.to, b2), ! {upon} (b2,b3)
```

but using the composition rules defined in Chapter 4, gives

{upon, overlap} * {upon} = {upon, overlap}

So that we can derive

! {upon, overlap} (a.from,b1), ! {upon, overlap} (a.to, b3)

which is exactly the same as

! link(b1,b3,a)

We can prove similarly that

! link(b1,b2,a), ! dropBox(im,b1,b3) ! link(b3,b2,a)

Thus, following dropBox, any links which were attached to the first box become attached to the second. The significance of this proof is that it shows how to achieve an analogue of the domain model action mergeNodes, by exploiting the spatial properties of the chosen representation. A complex action on a structure has been represented by the telling of a single spatial constraint: the effects of the action are implied by the composition properties of the system of spatial relations.

We will sometimes wish to remove all constraints on a box. This will be of use when we wish to drop a set of arcs. The following agent achieves this by creating a new box with the same identity as the old. We have already seen this technique in a different context when we considered the agents freeHead and freeTail, although in those cases, the arrows do not have persistent identity, so we do not need to update the object list. We maintain a list of changed objects (delta). This will be used to give persistent identity to boxes when we come define the link between the domain and interface models. We are not going to map arrows onto domain objects, so they do not need the same treatment. These points are examined further in the next section (section 6.4.3).

```
deconstrainBox (im, im', b, b', delta) ::
        ! im'.boxes = im.boxes \ b + b',
        ! delta = {(b,b')}
```

Finally, we can describe the behaviour of the interface model thus

```
graphEditorInterface(im, im', delta) ::
        (graphEditorIM(im), graphEditorIM(im')),
        createBox(im, im', b) : deleteBox(im, im', b) :
        createArrow(im, im', b) : deleteArrow(im, im', b) :
        fixHead(im, a, n) : fixTail(im, a, n) :
        freeHead(im, a, n): freeTail(im, a, n) :
        (dropBox(im, b', b1), deconstrainBox(im, im', b, b', delta)) :
        (dropBox(im, b, b1), deleteBox(im, im', b))
```

The last two lines need some explanation. The line

(dropBox(im, b', b1), deconstrainBox(im, im', b, b', delta))

says that a box can recur in an unconstrained form and then be constrained to be upon another box. This will get interpreted as a *gesture* signifying that the node represented by the first box is to be moved to that represented by the second. The line

(dropBox(im, b, b1), deleteBox(im, im', b))

constrains one box to be upon another and then deletes the first. We have already seen that this has the effect of transferring all arcs attached to the first box to the second. The difference between the two cases is interesting. In the former case, we will have to make an explicit link between an action in the interface and the corresponding action in the domain model. In the latter case, the domain action is implied by the inferential structure of the interface. There will be many cases like the latter where complex interactions can be represented in a simple way by making use of the inferential structure of the representation. We will return to these points in the next section when we show how to link domain model and interface model.

6.4.3 Linking domain and interface models

We have thus far given entirely separate descriptions of domain and interface models. It is true that many of the actions of one model have easily recognizable analogues in the other, but neither model accesses information contained within the other. We now consider how to 'glue' together the two models. This will give us a partial description of an application: it is partial because we have not yet considered the behaviour of the interface at the level of individual displays, which will be the subject of subsequent sections.

Chapter 1 described a view of the user interface as a form of visual representation. In that chapter, we discussed the importance of the nature of the mapping between represented and representing worlds. We referred to work which suggests that a one-to-one mapping between objects in the represented and representing worlds is preferable from the user's point of view. Here we shall show how to use the notation in order to describe such a mapping between domain and interface models.

The essential requirement is to associate objects in the domain model with objects in the interface model. So, informally, we want there to be a box in the interface model for every node in the domain model; and we want there to be a

link structure in the interface model for every arc in the domain model. Moreover, the mapping should be described in a way that enables it to remain consistent whenever changes occur in either of the models. In particular, we need to guarantee persistence of reference: when an object in the interface model is deconstrained (see for example the discussion in section 6.4.2 concerning the agents freeHead and freeTail), we need to ensure that the replacement is mapped onto the same object in the domain model.

In the case of arcs and links, the mapping is derived from the identity of the boxes and nodes. That is to say, a link is not an object with persistent identity in the interface model, rather its identity depends on the nodes which its component boxes represent. If one of the ends of the arrow is deconstrained and attached to a different node, the link no longer represents the same arc. On the other hand, if we consider the case of a box which recurs, then we want the new version of the box to represent the same node in the graph as did the old version.

There may be cases where an action in the domain model can only be represented in the interface model by a complex sequence of interface actions. In such cases, we may wish to initiate the action by a gesture, that is to say an action in the interface model which is used to trigger an action in the domain model. In such cases, the representation is not *homomorphic* in the sense in which we defined the term in section 1.1.1.

Most of the linkage between domain and interface models can be achieved by mappings between nodes and box ids and between link structures and arcs. The mapping between boxes and nodes is as follows.

```
graphEditorMap(im, dm, boxMap, linkMap) ::
links ^
        ! boxMap = dm.nodes @ im.boxes,
        ! linkMap = dm.arcs @ links,
        collect a : im.arrows into l : links [
                any b1: im.boxes [
                any b2 : im.boxes [? link (b1,b2,a)]]],
        all (arc, (b1,b2,a)) : linkMap [
                ! includes(boxMap,(arc.from, b1)),
                ! includes(boxMap,(arc.to, b2)) ]
```

The set links is not defined explicitly in the interface model: it is *derived* from the properties of objects in the interface model. The collect expression defines the membership of this set and the all expression says that both boxes in a link must map onto the corresponding nodes in the arc which the link represents.

The definition is not quite complete, because we also need to ensure that the mapping between nodes and boxes remains the same for those boxes and nodes which are not removed. Also we must take account of what happens when a box is deconstrained: we need to ensure that the box mapping preserves the reference between the node and the updated version of the box. We achieve this by adding constraints to delta (the set of updated regions).

```
DILinkage1(im, im', dm, dm', delta) ::
boxMap, linkMap, boxMap', linkMap ^
        graphEditorMap(im, dm, boxMap, linkMap),
        graphEditorMap(im', dm', boxMap', linkMap'),
        all (r, r') : delta [node ^
                ! includes(boxMap,(node, r)),
                ! includes(boxMap',(node, r')) ],
        all (n,b) : boxMap [
                ? im'.boxes = im'.boxes + b -> ! boxMap' = boxMap' + {(n,b)}
                | ? dm'.nodes =  dm'.nodes + n ->
                        ! boxMap' = boxMap' + {(n,b)} ]
```

This agent ensures that whenever any box is deconstrained, the mapping between nodes and boxes is updated so that the node is mapped onto the new region in the next state of the domain model. Note that, although we also deconstrain arrows in order to swing them round to other nodes, we do not need to update their referents, because they are not mapped onto any objects in the domain model.

Finally, we complete the definition by considering the case where a node is to be moved. Remember that we represented this action by a gesture (see section 6.4.2). We therefore need to ensure that the appropriate domain action is taken.

```
DILinkage(im, im', dm, dm',delta) ::
n1, n2, b1, b1', b2, linkMap, boxMap ^
        graphEditorDM(dm, dm'),
        graphEditorIM(im, im'),
        graphEditorMap(im, dm, boxMap, linkMap),
        DILinkage1(im, im', dm, dm'),
        ( (?dropBox(im, b1', b2), ?deconstrainBox(im, im', b1, b1', delta)) ->
        moveNodes (dm, dm', n1, n2),
        ! includes(boxMap,(n1, b1)),
        ! includes(boxMap,(n2, b2))
```

So in this case, all we have done is to check for the completion of the gesture (which involves deconstraining a box and dropping it onto another box), and then we tell explicitly which action to take in the domain model. The mapping between other interface actions and domain actions is already implicit in the mappings between

objects. We have discussed one example of this already, when we showed that

(dropBox(im, b, b1), deleteBox(im, im', b))

had the effect of transferring all arrows from one box to another. Because of the mapping between links and arcs, this will have the effect in the domain model of moving arcs from one node to another.

6.4.4 The presentation component

We now consider changes which occur at the presentation level. For example, we may wish to model operations which allow the user to reorganize the layout of the graph: such operations do not have any semantic consequences, though they are clearly important from the point of view of usability. Additionally, there are operations on the interface which do not have any direct counterpart in the domain model, but which represent intermediate states in domain operations. An example of this is the creation and manipulation of arrows: an arrow on its own has no counterpart in the domain, but when constrained to connect two nodes, it represents a directed arc between the two nodes in the graph.

We will model the user input as a set of agents executing in parallel to the domain model and interface model agents. Operations at the presentation level define constraints between successive instantiations of the interface model. We will define a presentation model, and describe dynamic behaviour as a sequence of models. The linkage between presentation and interface models will be such that each occurrence of a presentation model is an instantiation of an interface model. The presentation model therefore is a mapping between objects in the interface model and instantiations of those objects. We have already seen how such generic interactors may be defined for operations such as movement, and applying constraints between objects.

Constraints between successive states of the presentation model describe how the display state responds to changes brought about by actions such as dragging. These issues were considered in section 6.3.2 when we showed how movement could be represented using agents. We remarked in that section that there will generally be a number of different, spatially consistent ways in which a given set of constraints may be instantiated following a change and that (paradoxically) dynamic behaviour is largely characterized by the set of objects which remain unchanged. The 'principle of least surprise' may serve as a guide to the designer, but it is too ill-defined and loaded a term to be capable of direct translation into constraints. Therefore it will often be necessary for us to define explicitly which

objects may move following a given input and which remain static.

Operations on the display relate to individual regions in a specific instantiation of the interface model. The display is not structured explicitly, but it has an implicit structure in the interface model. The operations which we are about to define will act on an unstructured set of regions, but this set will be mapped to the interface model which it instantiates. Thus we will be able to identify regions in terms of the part of the interface model which they instantiate.

One of the possible operations is to move a node. When a node is moved, we want other nodes to remain fixed. The arrows attached to the moved node may move as a consequence, because of the constraints that attach them to other nodes. We define this behaviour as follows

```
moveBox(im, pm, pm', box) ::
connected, newArrows, oldArrows, b, b' ^
        ! includes(pm,(box, b)),
        collect a:im.arrows into a:connected [? touch(a.from, box) :
                ? touch(a.to, box)],
        ! move(b, b'),
        replaceAll(pm,pm', (connected @ newArrows) + (box @ b'))
```

In the new map pm' the moved box is no longer associated with the instantiation b, but with b'. Notice also that we have made no change to the box itself, nor to the constraints upon it, only the instantiation of the box has been moved. Similarly, we have updated the instantiations of the arrows attached to the box, so that they can move together with the box, but we have made no change to the actual arrows in the interface model.

We can also see from this example that we use the interface model to carry structural information: the presentation model is a mapping, and as such contains no structural information; it is just a set of paired region values. The advantage of this approach is that we may define operations on the presentation model that are entirely generic, since they do not presume any particular structure in the model.

Movement of individual arrows is of two kinds: the arrow may be swung around, keeping one end fixed, or it may be translated. Our definition of move produces a rigid translation, because the constraints between parts of the moved object are kept fixed. However, we can achieve the effect of swinging an arrow by the head or by the tail with the following agents.

```
swingHead(im, pm, pm', arrow) ::
```

```
a, a' ^
        ! includes(pm, (arrow @ a)),
        ! move(a.to, a'.to),
        ! a.from = a'.from,
        ! replace(pm,pm',arrow , a')
swingTail(im, pm, pm', arrow) ::
        ! includes(pm, (arrow @ a)),
        ! move(a.from, a'.from),
        ! a.to = a'.to,
        ! replace(pm,pm',arrow , a')
```

The definition of a non-rotating translation for arrows is similar to that for nodes, except we will define it so that only the arrow is moved, effectively detaching the arrow from any linked nodes.

```
moveArrow(im, pm, pm', arrow) ::
        ! includes(pm, (arrow @ a)),
        ! move(a, a'),
        ! replace(pm,pm',arrow , a')
```

We will restrict the possibilities for movement in the display state to those changes which can be described by the four agents moveNode, moveArrow, swingHead and swingTail. There are of course many other possibilities which we could include. For example, we might wish to move whole subgraphs in a single operation. For any application, it will be a design choice as to which forms of movement are to be permitted in the display: our aim here is to show how the agent notation may be used to define different kinds of movement, rather than to produce a definitive design for graph editor applications. We do not make any claims regarding the usability of the design.

The presentation agent is the disjunctionof the four agents defined above.

```
graphEditorPresentation(im, pm, pm') ::
a, b ^
        moveArrow(im, pm, pm',a) : moveBox(im, pm, pm',b):
        swingHead(im, pm, pm',a) : swingTail(im, pm, pm',a)
```

We also need to define the linkage between the presentation and the interface model. The presentation model is a mapping from the interface model, so that successive presentation models are instantiations of the interface model. However, we need to take account of the possibility that the interface model may get updated, in which case the presentation model should be an instantiation of the updated interface model.

```
PILinkage(im, im', pm, pm') ::
```

```
inst, inst' ^
            ! pm = im @ inst ,
            ( ! pm' = im' @ inst': ! pm' = im @ inst' )
```

6.5 Conclusions

In this chapter we have shown how the agent notation of Chapter 5 may be used in the specification of user interfaces. A main theme of the chapter has been to demonstrate that the notation allows the specification of the interface as a set of components broadly following those of the Arch Model which was described in section 2.2.3.

The model which we have proposed for the user interface architecture, as outlined in the previous chapter (section 5.7) and in the introduction to this thesis, has three major components:

- a domain model in which domain objects and relations are modelled by structures;
- an interface model which describes the set of possible display states, modelled using spatial and constraints between geoms;
- a display model which describes dynamic behaviour of the presentation using constraints between successive display states.

We have additionally described how these three components are related to each other. There is a one-to-one mapping between structures in the domain model and a subset of geoms in the interface model. Not all of the geoms in the interface model have direct significance to the domain model. Some of them represent incomplete actions. In the example we developed, only arrows which were constrained to touch boxes had significance in the domain model.

We distinguished between sets of objects which were basic elements of the interface and defined extensionally (i.e. the set membership is determined by explicit enumeration of members) and those derived sets which are defined intensionally (by possession of a property). Some of the significant geoms in the interface model are defined intensionally, by reference to other sets of geoms in the interface model. For example, the link geom is defined in terms of the set of boxes and arrows. In general, such sets defined intensionally will represent *relations* in the domain model. For example, the arcs of the graph editor are a relation between nodes. The definition of the interface model includes constraints which maintain the consistency of sets which are defined intensionally. In order to add a new element to such a set it is sufficient to apply constraints to one of the basic sets. The implication of this is that the process of defining relations between

objects in the domain model can be described as a series of generic interactions involving the application of constraints to arbitrary regions in the interface model.

The linkage between display and interface models is one of instantiation. Every display state has the same structure as the interface state and additionally it has a ground constraint net in which exactly one of the spatial relations applies between any pair of objects. Successive display states must not only conform to the constraints in the interface model, but also to additional constraints which describe the dynamic behaviour. We showed how various forms of movement could be represented using qualitative constraints applied between successive display states. The interaction components employ the linkage between display and instantiation to add constraints to the interface model based on the current state of the display. With gestures such as drag and drop, the instantaneous state of the display may determine an action on the interface model. The interactors which we have defined are truly generic, in that the actions they perform on the interface model do not rely on knowledge of the structure of that model.

Chapter 7 The research in context

In this chapter we compare the approach taken in this research with other approaches to the problems which have been addressed and suggest some possibilities for further research. In section 7.1 we summarize the argument of the thesis. Then in section 7.2 we make a comparison with other research into the use of constraints in UIMS. We make a comparison between the thesis and models of UIMS in section 7.3 and finally in section 7.4 we suggest possibilities for future research.

7.1 Introduction

This thesis proposes that the concurrent constraint programming framework, cc(), if instantiated with an appropriate system of constraints, has wide areas of application to the specification and implementation of graphical user interfaces. To support this proposition, we have pursued three goals:

- to develop a suitable system of constraints and show how it may be integrated into the framework;
- to investigate the relevance of the framework to different components of the user interface;
- and to provide examples of the use of a notation, based on the framework, which we have developed specifically for use in user interface construction.

In pursuit of the first of these goals, we argued for the use of specialized constraint systems and have described a system of constraints based on the use of qualitative spatial relations. We maintain that such a system of constraints has particular advantages over more general constraint solvers, such as those based on the solution of real-number equations. These more general systems do not have effective abstractions for describing the relationship between graphical objects in the interface and the underlying functionality which they represent: even quite simple spatial relationships may be awkward or impossible to describe.

Throughout the thesis we have used the arch model as a reference for the

second goal and have shown how to model its components using constraints. The presence of a synchronization primitive, in the from of blocking ask, makes the cc() framework suitable for modelling aspects of dialogue; the spatial constraint system gives a compact means for describing presentation aspects of the interface; and the ability to model sets of structured objects by means of constraints on geoms allows us to construct a domain model.

Finally, in pursuit of the third goal, we have shown how typical idioms of interaction may be described using the notation and have given an extended example to show how an interface specification may be structured according to the arch model.

A unifying theme has linked the exposition of these goals. This is the idea that the user interface may be considered as a form of visual representation. It raised a number of questions which we have addressed at length in the preceding chapters.

First we considered the nature of representations in general. This raised the questions of 'What is a representation ?' and 'How can representations be characterized ?'. The research on these questions shows a strong consensus that the complexity of the correspondence between represented and representing worlds is a determining factor in the usability of representations. In particular, several authors have independently given characterizations of a form of correspondence which is essentially a one-to-one mapping (section 1.1.1). Stenning and Oberlander term this a 'MARS' [138], Barwise and Etchemendy term it a 'homomorphic representation' [7] and Levesque has referred to it as 'vivid knowledge' [95]. All conclude that reasoning is easier with such representations, even though there may be some loss in expressiveness compared with representations which do not have such a simple correspondence between represented and representing worlds.

This led us to ask whether some form of homomorphic mapping might serve as a basis for specifying the correspondence between the display state of the user interface and the domain model which it represents. We concluded that an attempt to do this directly was doomed to failure because of the high degree of redundancy and ambiguity in typical graphical user interfaces. Basically, these problems stem from the dynamic nature of the interface. In a static form of representation, such as a diagram, information about object relations is carried by properties of objects, such as size and colour, and relations between them such as

alignment. In the user interface, information is additionally carried by the dynamic behaviour of objects. This means that there will generally be many possible displays which correspond to a given domain model state (redundancy) and a given display state may correspond to more than one domain state (ambiguity). We gave an example of this in Figure 1.1.

However, if we choose to consider the set of possible display states, rather than a single display state, then the situation is different. In this view, the user explores the interface state by manipulating objects in order to discover what relations are possible between them. Relations between domain objects are not mapped directly onto single visual relations, but onto sets of visual relations. Information about the domain is carried by the *possible* relations between visual objects as well as the *actual* ones which hold at any instant.

In the context of specifying and implementing systems, this view has several implications: it requires a way to represent the sets of allowed states of the display; there must be a mechanism to handle communication between domain and interface state; the view needs to be related to existing models of the user interface so that it can be incorporated into established methodologies for user interface development; above all, we need to have some notation for describing the correspondence between represented and representing worlds. The choice of the concurrent constraint programming paradigm as the underlying computational model goes a long way to meeting these requirements. The essence of constraint programming is reasoning with sets of allowed states. Therefore, it is a natural way of representing the interface state as a set of allowed display states.

A strong feature of the cc() paradigm is that it naturally expresses the temporal ordering of interface dialogues. Concurrent constraint programming provides a synchronization primitive, blocking ask, which can be used to handle interprocess communication. In this thesis, we have incorporated the basic cc() framework and systems of constraints specifically designed for description of spatial relations and picture structure. This means that the same notation can be used to describe presentation-specific detail and dialogue structure.

We have aimed at a level of description of the interface which is sufficiently abstract to support interface specification and formal reasoning while at the same time being based upon a model of computation that affords the possibility of implementing the description in a more or less direct manner.

We have shown that the set of basic relations is logically complete in the sense that any set of 2.5D regions has a unique description in terms of the basic relations. We have also seen how the basic set may be elaborated to produce alternative systems with finer distinctions, such as edge contact. Our choice of a particular system has been based largely on pragmatic considerations: the system has to be computationally manageable and sufficiently rich to describe an 'interesting' set of interfaces. We demonstrated the latter by showing that it could describe some common idioms in direct manipulation user interfaces. In taking this approach, we have avoided the question of what constitutes a 'good' representation: it might or might not be the case that the interfaces we have described conform to principles of software ergonomics; all we can say is that the interfaces which have been described are typical of those in widespread use at the time of writing.

7.2 Constraint-based UIMS

In this thesis we have presented a description of UIMS in which constraint programming has a central role. Other researchers have proposed using constraints in UIMS. In this section we summarize the main differences between these other approaches and the one advocated in this thesis. We will use Garnet [109, 109] and Rendezvous [75, 114] as examples, since both of these are relatively mature and well-researched systems which address the major issues in UIMS research.

We will consider five different aspects of the use of constraints in which our approach differs fundamentally from that used by other researchers. In section 7.2.1 we argue that we are proposing a new role for constraint programming in the UIMS architecture, based on the idea that constraints can be used as the primary medium for information, rather than being just a mechanism for maintaining dependencies between informational objects. The next section (7.2.2) considers the importance of the property of stability in constraint systems. We argue that where this property is lacking, then the constraints do not have a truly declarative semantics. In section 7.2.3 we look at how change to the constraint net is handled in different systems. An important issue in the handling of change is how structural changes in the domain model are kept consistent with the presentation and we consider this in detail in section 7.2.4. Finally in section 7.2.5 we consider generic interactors.

7.2.1 Constraints as information

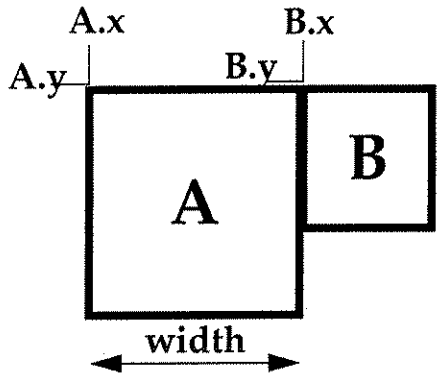
Other research into constraint programming for UIMS has taken an approach in which information is contained in *values*. Constraints have a secondary role as a means to express dependencies between values. In this view, the central issue is to find some individual solution to the system of constraints. If there are multiple solutions, this is either treated as an error or as a problem to be resolved by an arbitrary choice of solution. One of the reasons for the prevalence of this view of constraints is that the original motivation for using constraint satisfaction techniques in UIMS was as a means to solve layout problems. With such problems, the goal is to discover any solution for a given system of constraints.

By contrast, our motivation has been to express the semantics of visual representations. The central issue for us is not to find specific solutions, but to check if a given set of constraints is consistent. We use a computational model in which constraints, rather than values, are the primary medium for information. We are not concerned with specific values, but with the possible relations between values, and we express these using constraints. Solution of the constraints has a secondary role: we require a mechanism to check the consistency of the constraint store following a constraint tell, but do not require it to provide individual solutions. Rather than eliminating multiple solutions, we use them to increase the expressiveness of the model, since they allow us to represent the interface state as a set of possible display states.

Our approach has been based on qualitative reasoning about systems of spatial relations. This enables us to express incomplete knowledge about the spatial relations between objects. The approach taken by other research on constraint programming in user interface construction is to use general purpose quantitative systems in which spatial relations may be described as special cases, typically by imposing numerical constraints on points in the 2D plane. In these systems incomplete knowledge may be expressed as inequations in which values are constrained to lie in a range. However, solution of such constraints is handled inadequately or not at all by systems based on local propagation (for a review of local propagation, see section 3.2.1).

Rendezvous uses one-way constraints, with explicitly identified source and target variables. In Figure 7.1 the solution to a simple layout problem is shown. This example is taken from Hill [75]. The problem is to maintain the box **B** aligned with the right-hand edge of box **A**. The constraints are explicitly one-directional,

with designated source and target variables. This means that, in the example, changes to the position of box **A** cause changes to the position of **B**, in order to maintain the desired adjacency relationship, whereas the converse is not true.



sources: A.x, A.width
targets: B.x
method: +

sources: A.y
targets: B.y
method: identity

Figure 7.1 A simple layout problem and its solution by constraints in Rendezvous. The definition for the two relevant constraints is shown alongside.

The constraints are defined at a low-level of abstraction, detailing the implementation of the constraint (sources, targets and methods). This increases the range of constructs that can be expressed, but it also makes the behaviour of the system harder to understand since the constraints are not expressed in a declarative manner. A general problem with this approach is that in order to predict and reason about the behaviour of the interface it is necessary to have a detailed knowledge of the implementation. Two-way constraints in Rendezvous have to be expressed explicitly, by declaring constraints that define the inverse relationships. The constraint satisfaction mechanism also requires these to be declared explicitly as inverses (in order to prevent cyclic reevaluation of the constraint and its inverses).

A sample of some Garnet code and the widget it produces is shown in Figure 7.2. This is a slightly modified version of an example in the Garnet reference manual [109]. Although simple, the example illustrates some important issues with using constraints to program the user interface.

The behaviour of the object depends on constraints that are hidden from view. In Garnet, the prototypical graphic object can have its bounding box set directly by the designer using the slots top:, left:, height: and width:. The position of the whole widget is set by the code in line 2. However, compound objects (termed 'aggregates' in Garnet) have additional constraints that cause the height and width of the aggregate to be calculated from the subcomponents. So in this case it

```
[1] create-instance TEXT-LIST opal: aggrelist
[2] :left 0 :top 0
[3] :items ("One" "Two" "Three" "Four")
[4] :direction :vertical
[5] :fixed-width-p T
[6] :item-prototype
[7] opal: aggregadget
[8] :parts
[9] :frame opal:rectangle
[10] :left (o-formula (gv l:parent :left))
[11] :top (o-formula (gvl: parent :top))
[12] :width (o-formula
[13] (+ (gvl :parent :text :width) 4)
[14] :height (o-formula
[15] (+ (gvl :parent :text :height) 4)
[16] :text opal: cursor-text
[17] :left (o-formula (+ (gvl :parent :left) 2)
[18] :top (o-formula (+ (gvl: parent :top) 2)
[19] :string (o-formula (nth
[20] (gvl :parent :rank)
[21]  (gvl :parent :parent :items)))
```

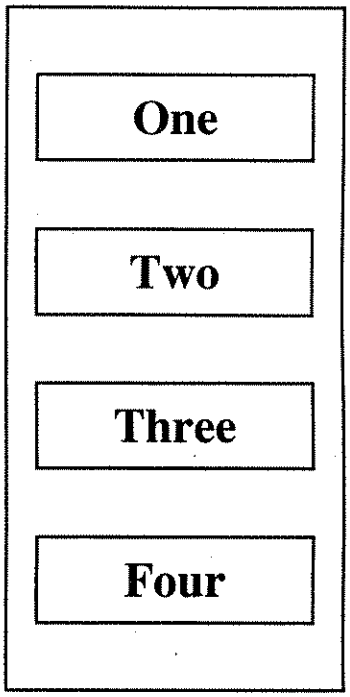


Figure 7.2 Some Garnet code and the widget it produces. The line numbers are for reference only, and not part of the code.

would be an error for the designer to set height and width explicitly. It would be possible to rewrite the constraints on the aggregate so that the size of the subcomponents was determined from the size of the whole object, but then it would be an error to set the size of the subcomponents explicitly. This is a consequence of the use of one-way constraints and reveals a general problem with their use: the behaviour of an object will depend on the implementation of the constraints, so that the advantages of the declarative nature of constraint programming are diminished. By contrast, the qualitative spatial constraints which we use are fully multidirectional.

The use of local propagation also restricts the range of problems that can be

solved. Hill claims that this is not unduly restrictive:

> ... because local propagation considers very little information in each step of the propagation it is limited in the range of constraint problems it can solve. However, experience has shown that it is adequate for most user interface tasks. In addition, local propagation is easy to implement and gives predictably fast performance (often linear in the number of constraints). [75, p226]

So what are the tasks for which it is suited? Local propagation is adequate for layout tasks in which the only spatial relations between objects are offsets whose values can be determined exactly as in the example in Figure 7.1. However, local propagation methods do not cope properly with indeterminate values, such as might occur (say) if the task in the example were to assert that box B is contained anywhere within box A. To constrain a value to a range using local propagation methods requires two representations of the value: one is written, the other is read. This is shown in Figure 7.3.

sources: writeValue, minValue
targets: valueOrMinimum
method: max

sources: valueOrMinimum, maxValue
targets: readValue
method: min

Figure 7.3 Constraining a value to lie within a range using local propagation in the style of Rendezvous. The variable **writeValue** is written by other constraints, but the constrained value has to be read back from another variable **readValue**.

This is unsatisfactory for a number of reasons. The constraint does not actually generate a value for the variable until one is supplied from somewhere else. Compare this with the boxes example where once a value had been supplied for the position of box A, the position of B could be determined. If it was required to constrain the corner of B to lie within A by applying a range constraint to the x and y values of B, then even when the minimum and maximum values of the range had been determined from the coordinates of box A, neither of the methods in the range constraint could be activated. Furthermore, variables which are to be constrained in this way have to be treated differently from others where the value

can be read or written from the same variable. Worse, it does not work correctly where there are multiple range constraints on the same variable. The reason for this is that local propagation does not support indeterminate values: a value is either known completely or not at all. In the case where several constraints write different values to the same variable, one of the values has to be chosen arbitrarily. Techniques based on propagation of intervals (sometimes referred to as *tolerances*), are known and have been described in the AI literature, but have not been used in UIMS work [78].

An alternative approach to expressing inequalities in taken in Garnet. Here, wherever an inequality occurs it is expressed as an equation including an arbitrary constant. e.g.

$x < y$

is expressed as

$x + c = y$

where the constant c is some arbitrary value (typically 1). This is clearly unsatisfactory as a means to express indeterminate values. Note that this is not a variation on Simplex, such as is used in CLP(R) (see 3.3.1 and [81, 82]), because where there are several such constraints, no attempt is made to find a consistent set of valuations for the constants.

Of course, this kind of partial information is easily expressed using qualitative constraints.

In the notation which we have presented, disjunctions allow us to express conditional execution of agents. Some other systems allow conditional execution based on comparison of values. For example, in Garnet, the use of demons allows us to make some operation conditional on two values being equal. However, in Garnet an operation cannot be conditional on two values being *constrained* to be equal. This is because in Garnet the constraint mechanism does not support reasoning about multiple solutions, so that we cannot make a distinction between valuations which are necessarily entailed by the given constraints and those which are permitted, but not entailed. We have already argued strongly that without the ability to make this distinction there is a problem of distinguishing ambiguous representations.

Moreover, in Garnet, the demon mechanism and the constraint mechanism are separate and require separate specification.

7.2.2 Stability

The approach to modelling change that has been taken in other work on the application of constraints to user interface construction is to allow the retraction of constraints. In the cc framework constraints may not be retracted once they have been told, therefore it is necessary to use the recurrence of agents as a means to represent change of state. Although this complicates the description of dynamic behaviour, because some changes of state have to be modelled by the introduction of new variables (we shall consider this in more detail in section 7.2.3), it also ensures that the basic operations of ask and tell are stable. Stability is an important property of constraint systems as it makes the behaviour of the system independent of the order in which the individual constraints are satisfied. Steele states the importance of order-independence as follows

> ... principles of order-independence, locality and monotonicity ... are essential to the design of a constraint system. Order-independence prevents the system from relying on the form of the input; its computations should be derived from the content only. Locality and monotonicity are important to the conceptual simplicity and comprehensibility of the system. [137, p364]

Many implementations of local propagation do not possess order-independence. In Rendezvous, for example, where there are conflicting constraints, the final evaluation for a variable is determined in an arbitrary way by the implementation. The constraint solver is two stage, having a plan–evaluation cycle. During evaluation, cycles in the constraint graph are ignored, thus ensuring that in each evaluation cycle any variable is given a value at most once. This contrasts with other forms of eager evaluation (e.g. the Hoover algorithm used in Garnet), in which the planning stage orders the constraints to avoid repeated evaluation. Hill refers to these limitations in his discussion of the constraint mechanism of Rendezvous.

> In general the value of a variable is determined by the last constraint evaluated that has the variable as a target. This value may not be consistent with other constraints that have the variable as a target ...
>
> Still we believe that it would be best to base future constraint systems on solvers that have support for multi-way constraints and cycles, but only use elaborate (and time-consuming) solution techniques on the rare occasions when they are needed. [75]

A major problem with this approach is that it is difficult to reason about the indeterminism which it introduces. The underlying computational model in Rendezvous is not order-independent: the effects of the constraints depend on the order

in which they are evaluated. This contrasts with the semantics defined in this thesis in which ask and tell are stable, so that the effect of telling a set of constraints is independent of the order in which they are evaluated. This form of stability depends on there being a notion of *failure* of constraints: there has to be some way to deal with situations in which two inconsistent constraints are applied. In our approach, whenever one of a set of (parallel) constraints fails, the whole conjunction fails. The notion of failure is explicit in the semantics of the constraints and can be exploited through the use of disjunction. Rendezvous, in common with other constraint-based UIMS has no explicit notion of failure so that any behaviour conditional on failure has to be implemented in a one-off way. This reveals a bias toward implementation efficiency at the expense of a well-founded semantics for the constraint system.

An important difference between the approach which has been outlined in this thesis and other constraint-based approaches to user-interface construction is the use of ask to control the ordering of constraint satisfaction. In our notation, ask allows us to suspend constraint tells until specific information is available. This ability is explicit in the notation. In systems based on dataflow constraints, such as Garnet, this kind of control exists, but is implicit within the constraint satisfaction mechanism. In such systems, constraints are only activated when there is sufficient information available to activate one of the methods associated with the constraint. The cc() framework allows such dependencies to be made explicit by the use of the ask operator. This effectively introduces a synchronization operator which can be used to express process-oriented properties of the interface, such as dialogue sequencing.

Other systems handle this issue with some kind of static analysis of the structure of the constraint net. For example, early versions of Garnet simply prohibit more than one constraint to write to any given variable. If there is only one constraint which can constrain any given variable, then the network of constraints can never become inconsistent. Later versions of Garnet use an algorithm called SkyBlue [123] to handle multiway constraints. This uses constraint hierarchies as a means to resolve conflicts (see section 3.2.1).

7.2.3 Dynamic changes to the constraint store

Hill claims three innovations in Rendezvous:

- indirectly referenced source and target variables;
- tolerance of side effects;
- special handling for variable initialization.

Indirect referencing is also used in Garnet. With indirect referencing the objects governed by a constraint may be changed dynamically at run-time. This is clearly necessary in a UIMS where objects are created and destroyed dynamically and constraints have to be applied to them. However, it means that the evaluation mechanism has to take into account the possibility that the constraint graph will be changed during evaluation: change to a value in the graph may result in the creation or deletion of new objects. Rendezvous handles this possibility by deferring the evaluation of constraints which modify the graph until all direct constraints have been evaluated.

The use of indirection in constraint systems based on local propagation requires an additional component in the dependency maintenance mechanism, since we have to allow for changes to the reference as well as changes to the referenced value. Automatic update of values occurs whenever source or target objects change as well as when source values change; object references need to be updated before values, so as to avoid assigning values to objects that will be removed or changed subsequently in the same cycle. In Rendezvous, updating of object references is by means of higher-order constraints that rewrite the source and target lists of other constraints.

Side effects are effectively constraints without targets in which the method part has some effect on the underlying system. In Rendezvous, side effects are used to communicate with the functional core. Side effects may be potentially harmful if they cause the constraint graph to be rewritten (as would happen if new objects were created by the side effect), so the programmer may cause them to be deferred in the same way as indirect constraints, by calling a special function deferSideEffect.

We saw in the discussion of stability in section 7.2.2 why this kind of behaviour is undesirable. The use of side effects means that properties of order-indeterminism are declarative transparency cannot be guaranteed. Moreover, update of the constraint graph is achieved through operations in the underlying implementation language, so that the dynamic behaviour of the constraint system has to be expressed separately from the constraints themselves. By contrast, in our notation update of the constraint graph is achieved through operators in the

notation itself, which have a well-defined operational semantics.

The constraint system in Garnet is based on a knowledge representation language called KR [56], which is implemented in Common Lisp. KR is frame-based and has user-defined inheritance and relations, combining object-oriented programming, constraint maintenance and knowledge representation, with inheritance defined using the prototype–instance paradigm, rather than the class–instance paradigm. Constraint maintenance is through one-way constraints, called formulae, which are installed in slots in object schemata. These reference slots in other schemata, using either global or local names. Indirect reference is achieved through dynamic look-up of slot names at run-time. This gives rise to a similar set of problems as in Rendezvous, whereby it is necessary to update the structure of the constraint graph using a separate mechanism from that which updates the values.

Two methods are available for propagation of change: value propagation and demon evaluation. Lazy evaluation is used for value propagation, though an eager algorithm is being tested. These algorithms are discussed below. Values, once calculated, are cached so that the formula does not have to be recalculated each time the value is required. Demons allow the implementation of side-effects. A demon is a piece of functional core code attached to a schema. In Garnet demons are used to provide control over aspects of change.

The algorithms used in Garnet are described by Vander Zanden *et al* [164]. The basic algorithm is a form of lazy evaluation in which constraints keep lists of dependents (other formulae which access the value returned by a formula). Whenever a value is changed, its dependents are marked recursively as invalid. If an invalid value is accessed then the formula associated with that value is evaluated, recursively evaluating any other invalid slots encountered. Structural change is handled by time-stamping the dependents lists, so that the current constraint graph is used to access values required by formulae. Out of date dependencies are removed while evaluation occurs.

An alternative algorithm using eager evaluation is proposed for Garnet. It is anticipated that this will yield faster evaluation. The choice between using eager or lazy evaluation depends on how many times changed values are accessed. If most changes to values result in changes to the display, then the overheads of marking values as invalid will not be compensated by savings in unnecessary recalculation. On the other hand, if changed values are not always accessed, then

the cost of recalculating them unnecessarily makes lazy evaluation preferable to eager. In practice, it appears that the former case is more likely and that most changed values get accessed eventually, so an eager evaluator is preferable.

The algorithm proposed for use in Garnet is a modified form of one described by Hoover [77]. It is a two-stage algorithm in which a planning stage is followed by evaluation. Constraints are given a priority ordering based on their dependencies, so that each constraint precedes its dependents in the ordering. The priorities are updated incrementally as the constraint graph is changed. Whenever a value is altered, its immediate dependents are added to a priority-ordered list and the evaluation stage consists of repeatedly removing the lowest priority constraint from the list, evaluating it and updating the list. This avoids repeated evaluation of the same constraint.

7.2.4 Structural consistency

One of the limitations which Rendezvous shares with other systems is its handling of structural consistency. This refers to the transformation between the structural relationships of interface objects and application domain objects. Whereas local propagation can maintain attributes of specific domain objects consistent with attributes of their counterparts in the interface, it is not suited to maintaining the structures of interface and domain consistent. As Patterson *et al* put it:

> Structural consistency is a problem that arises because Rendezvous expresses its viewing transformations with constraints. If the underlying objects form a hierarchy that is mapped onto a hierarchy of interaction objects, then a problem arises when one or the other hierarchy changes by adding a child or removing a child. Presumably the other hierarchy should also either add or remove the corresponding child, but this relationship can be difficult to express with simple constraints.
>
> We view this issue as a research question. We do not doubt that structural consistency can be maintained within Rendezvous, but our goal is to formulate a simple, robust mechanism that will insulate programmers from this complexity in the future. [114 p326]

This is a fundamental issue in the view which we proposed in Chapter 1 in which the user interface is treated as a form of visual representation. One abstraction which this involved was that of the correspondence between represented and representing worlds. In order to implement this correspondence in any concrete system, it is essential to be able to maintain structural consistency. We have shown in

this thesis how an approach based on qualitative constraints embedded in a concurrent constraint programming language enables us to handle problems of structural consistency in a simple and direct manner.

In our approach, we have used structural constraints to represent the domain model. Structural constraints are those which describe how geoms are built up into compound structures. The recognition and identification of structures is an essential part of the interpretation of visual representations, both for the user and for the system. Geoms are the basic elements from which the representation is created and structures allow the expression of information about relationships between the objects represented by the geoms. Structural constraints are used to define a range of part-whole relationships. These represent both the state of the functional core and the structuring of presentation objects.

Rendezvous uses the underlying object system (an extension of Lisp, called Mel), as a means to model structure. Changes to structure are brought about by the use of side effects (see above, section 7.2.3). This means that structural consistency cannot be handled entirely within the constraint system of Rendezvous.

Garnet uses KR schemata to implement object structure. Graphical objects are implemented through a special system **OPAL** which provides basic behaviour and structure. Initialization of objects is through default values associated with slots through inheritance. The formulae inherited by slots may be overridden both at instance creation time and dynamically at run-time, which enables structural changes to be conditional upon constraint evaluation.

One of the strengths of Garnet is that it handles some aspects of structural consistency between presentation and functional core. In the example shown in Figure 7.2, the number of objects to be displayed is not coded explicitly and the same code can be used to produce a vertical list of any number of text strings. This is a powerful feature: without it the designer has to be able to say in advance how many objects will be included in the list. Systems which do not have this feature are restricted to aggregate objects where the number can be determined in advance, such as menus. The ability to apply constraints to objects which are created dynamically is very important and greatly increases the expressiveness of the system. However, the kinds of aggregate which Garnet can handle are limited to those which have been predefined in the underlying object system. It is not possible in Garnet to use the constraint system itself to define new aggregates

since there are no constraints that refer to structure. We have explicitly defined constraints on geom structure so that we can manipulate structure using the same notation as for specifying other constraints.

7.2.5 Generic interactors

The interaction model of Garnet is often cited as one of its major strengths [108]. We reviewed this in Chapter 2. The basic interactors of Garnet are defined in terms of operations on a common structure possessed by all graphical objects. Each interactor changes some of the real-valued parameters which determine the shape, size and location of graphical objects; changes to these values are propagated using dataflow constraints to the other objects in the system.

At the lowest level of abstraction, these changes are constraint applications to region-valued variables in the store. Interaction at this level is generic: domain-specific properties and structures are not modelled explicitly and consequently interaction techniques which operate at this level may be reused across all applications. This form of separation is used by Garnet's interactors. We have given examples of how our notation may be used to define generic interactors in section 6.3.

However, our model of interactors is different from that used in Garnet. In Garnet interactors are applied to individual objects and operate by changing specific values. In our model, rather than changing specific values in the objects on which they act, our interactors apply constraints to those values. Additionally, we can define interactors which act on a structured group of objects, by applying constraints between those objects (e.g. drag and drop as described in section 6.3.4). This gives us a rich and extensible set of interactors. By allowing arbitrary conjunctions of constraints the interaction model proposed here is extensible and more expressive than that of Garnet.

Our model of the display state includes transition rules for specifying how it may evolve. The display is modelled as a sequence of states, each of which is an instantiation of an interface model. This allows us to make a useful distinction between semantically significant properties of the interface as those which are necessary in the interface model, and surface properties, which are merely possible in a display state. This kind of distinction is not possible when the shared store contains only the current instantiation of the constraints, as is the case in systems such as Garnet. Within this model, generic interactors relate gestures (sequences of display states) to actions (constraint tells on the interface model). In

our model, we can define interactors which apply arbitrary constraints to pairs of objects, whereas in Garnet generic interaction is limited to changing values within individual objects.

7.3 Models of the user interface

In this section we compare our approach to modelling the relationship between domain and presentation components of the interface with others that have been proposed. We consider aspects of modelling the domain, dialogue and presentation components of the arch model.

7.3.1 Domain models

The relationship between interaction toolkit and functional core is rarely a total one-to-one mapping from functional objects to toolkit objects, from functional core attributes to graphical properties of toolkit objects and from object relations in the functional core to object relations in the toolkit. For this reason, it is necessary to have a domain model which abstracts away from the implementation details of the functional core. In many cases, the domain model contains objects and relations that are only implicit in the functional core.

The essence of the problem faced by the UI designer is to specify a *representation*. That is, a mapping between objects and relations in the domain model to objects and relations in the presentation model of the interface. In our approach, geoms and structural constraints are used to model the domain. Geoms provide structure for the underlying regions and constraints and offer a level of abstraction that is mid-way between the generic and the domain-specific. Change at this level is restricted to creation and destruction of geoms, together with abstractions of state change. These latter are defined transitions in the state of the geom which reduce to atomic operations on the underlying geometry. They provide a link between domain model and presentation layer.

The aspects of the application domain which need to be modelled are varied and complex: this point was discussed in section 2.4, but it is worth repeating it here. The application domain includes objects, properties, parts and relationships between them. Moreover, the model may change dynamically. The easiest interfaces to construct are those which represent the properties of a fixed set of domain objects; the hardest those which represent sets of domain objects whose number, properties, structure and interrelationships change dynamically.

We need to be able to represent composite structures composed of multiple domain objects and relations between domain objects. The approach taken in

recent work on model-based UIMS has been to extend the application domain model with specialized objects that fill the role of inter-object relations and sets of objects. However, as Kovacevic observes [88], this approach becomes unwieldy when there are overlapping part-hierarchies involving complex (i.e. non-binary) relations between objects. The basic problem is that it becomes necessary to create explicitly objects which model relations that are implicit in the domain.

7.3.2 Dialogue models

The approach used for representing dialogue structure in model-based systems (e.g UIDE, Humanoid [43, 88]) uses pre-and post-conditions to define the allowed sequences of actions and can model adequately the requirements for individual operations in the application domain to be valid. However, it has been observed that pre- and post-conditions may be unwieldy tools for modelling sequences of actions (dialogues) for which process-oriented descriptions are often better suited, albeit the two approaches have the same expressive power [1,140].

In our approach, the blocking ask construct allows us to give a process-oriented description of dialogue structure.

The RED-PiE model was discussed briefly in Chapter 2. It has some similarities to the view that has been presented in this thesis. The RED-PiE model has explicit components for display and system state and several properties which can be derived from the model relate to the correspondence between these two components. However, this correspondence is defined implicitly by describing both display state (Display) and domain model (Result) as separate functions of inputs (Program) and state (Effect). The simplicity of the basic PiE model has allowed the development of many variations which are specially adapted to describing the formal properties of particular aspects of interactive systems.

The PiE model embodies a view of interaction which is input-oriented. We have had very little to say about input, particularly low-level input such as key-presses and mouse-clicks. Moreover, we do not have a view of the interactive system as a closed world in which all change is initiated by the user. Instead we model interaction using concurrently executing agents. In our model, there is no ontological distinction between changes originated by the user and those originated by the functional core. Within agent definitions, input is abstracted by the operation of telling constraints. We assume that the user has some means to manipulate the objects visible on the screen and we are able to prescribe how they

will behave while being manipulated. We can even refer to low-level events when we wish, by using external agents. However, all such low-level events ultimately get transformed into constraints on the structures in the interface. The only effects on the functional core which we model explicitly are those which occur as a result of changes to the interface structure. This does not mean that our model fails when the functional core is changed by 'invisible' user actions (e.g. function key presses): such changes reflected in the interface structure as though they had originated in the functional core.

It is clear that the dialogue model must be able to express the creation and destruction of objects and the values of their attributes. However, it also needs to be able to handle arbitrary relations between domain objects and translate these into relations and objects in the presentation model. Some of these relations may be generic, such as partOf and isA, others may be specific to the application. Additionally, there may be semantic constraints on the network of relations (for example to forbid cycles in part–whole relations).

In our approach, we use the same notation to describe domain and presentation models. It is therefore possible to reason about correspondences between domain model and display state. One possibility that this affords is that of making inferences about the appropriateness of the presentation model for the given domain model. For example, we can decide whether all semantic relations in the domain model have corresponding relations in the display. A simple example involving the relation isA is to compute the transitive closure of the relation. i.e.

$$\forall x,y,z \bullet isA(x,y) \wedge isA(y,z) \Rightarrow isA(x,z)$$

If such a relation is to be represented directly by a spatial relation, then it should presumably be a transitive one such as upon. We require that all possible states of the domain be capable of representation within the display and that all display states (excluding those which represent incomplete actions) should represent a valid domain state.

The rules for maintaining consistency of the domain model may be more or less explicit in the interface design process. There are advantages in having such consistency rules made explicit: they may be used for purposes such as automatic help generation and to check the validity of parameters to calls on the functional core; additionally the information flow between domain model and functional core is reduced. On the other hand, the model becomes more complex and some

of the responsibility for computation in the functional core is passed over to the interface. The extreme case is to have all computation take place within the domain model, thereby obviating the need for the functional core. The ability to use the inferential structure of the spatial constraint system allows us to model some of the consistency rules of the domain model.

7.3.3 Presentation models

Our view of the user interface, as set out in Chapter 1, and further refined in section 5.7 and Chapter 6, models the interface as a set of allowed instantiations, each of which corresponds to an abstract display state. Constraint programming provided a means to model the set of admissible displays which constitute the interface state.

The parts of the abstract display actually visible to the user depend on depth-ordering of the components and the positioning of objects relative to display surfaces (windows). We showed in section 6.3.5 how visibility could be modelled in the notation which was developed in Chapter 5.

The domain model is represented using structural constraints on geoms and related to the interface model by a one-to-one mapping between domain model objects and a subset of the interface model objects. The motivation for the model is to specify the link between the presentation layer and the domain model layer as a simple correspondence.

One consequence of this view of the user interface is that there are two distinct kinds of interaction: some interactions change the display abstraction currently presented to the user; others change the interface state itself, that is to say, they change the set of admissible displays. Other authors have also commented on this difference: Took refers to surface and deep interactions [148]; various authors working within the framework of PiE models use the terms *passive* and *active* [34,66,122].

We also require the ability to describe intermediate states of the visual representation which correspond to partially completed interactions. During interactions, the interpretation of the visual representation may differ from the current state of the domain. For example, when a file icon is being dragged, the constraints which define the position of the icon relative to that of its parent directory may be relaxed to allow its movement over the directory tree. Sometimes this is shown by feedback in the form of a copy of the dragged icon, sometimes by movement of the icon itself. In either case the representation differs

from the 'quiescent' representation when no interaction is in progress: the rules which describe the semantics of the visual representation inbetween user interactions are different from those which apply during them. Therefore the state of the interface depends both on the current set of spatial relationships between objects in the visual representation and also the semantics which apply to that state.That is to say, the relationship between visual representation and application domain is generally moded, with each mode having a different set of constraints linking the state of the application domain to that of the representation.

The existence of modes of interaction implies that the interpretation of the representation is context sensitive. From the user's point of view, there is another form of context sensitivity related to ambiguities in the visual representation. At any instant, even with full knowledge of the rules of interpretation, there may be ambiguities which can only be resolved through further interaction. This property of interfaces, that the domain state (*Result* in the Red-PIE model) can eventually be determined unambiguously from the state of the display, has been termed *truthfulness* by Abowd. The PiE model and its derivations model interaction as traces of input and display events, so that the property of truthfulness depends on the idea of reachable states of the display. However, the way that the display state is modelled is very coarse-grained since it does not provide any abstractions for describing how information is presented in the display: all that can be said is that a certain piece of information is or is not displayed.

In our approach we have an explicit spatial model for the display state and can relate specific features of the display state to the state of the domain model. We have therefore been able to offer a much finer-grained model of the correspondence between domain and display than is possible in the RED-PiE and similar models.

The user action notation (UAN) [67, 69]offers a means to describe informally aspects of the visual structure of the display. However, the range of spatial relations which can be expressed is limited to those which describe the location of the mouse point with respect to graphical objects. The UAN is intended as a means to support communication in the design process and is therefore an informal notation. It does not have a well-founded semantics so that it is not possible to reason about UAN specifications. By contrast, we have offered a much richer set of visual relationships within a notation that has a well-founded semantics.

7.4 Future research

We conclude this chapter by suggesting some possible directions for future research based on the work reported in this thesis.

7.4.1 Implementation of the notation

We have given a formal description of the semantics of the notation and have frequently made reference to issues involved in implementing it as a programming language. Implementations of other languages within the cc() framework have been widely described in the literature (e.g. cc(fd) [70] and Janus [83, 127]). One of the advantages of working within the cc() framework is that the same basic mechanisms for handling synchronization can be reused with different constraint mechanisms for checking entailment and consistency. A number of 'skeleton' implementations of concurrent constraint programming languages are currently under investigation. For example, Haridi *et al* [65] propose a parameterized language based on the Andorra Kernel Language (AKL), and Fruhwirth describes a parameterized language using constraint-simplification rules (a form of rewriting) [54]. The essential component that must be added to such skeleton implementations is a mechanism for checking consistency and entailment of constraints. We described algorithms in Chapter 3 for checking the consistency of constraints. Although the general problem of checking the consistency of a labelled constraint graph requires exponential time, there are weaker forms of consistency that can be achieved in polynomial time. For many practical applications, such weaker forms of consistency may turn out to be perfectly adequate substitutes for full consistency. For example, the finite domain cc() language, cc(fd), uses path-consistency in the constraint-satisfaction mechanism [70].

7.4.2 An agent calculus

One of the most appealing features of languages in the cc() framework is that they offer a well-defined semantics for the synchronization of concurrently executing agents. An interesting possibility is to develop a calculus for cc() languages in the style of Milner's CCS [103] or Hoare's CSP [76]. This would allow proof of equivalence properties between agents. Saraswat describes some research along these lines toward developing a bisimulation semantics for cc() programs [126, Ch 10, 11]. This is based on the notion of *reactive equivalence*, in which the effect of an agent is defined in terms of the state of the constraint store. Two agents are reactively equivalent if in any given store, where one of the agents can reduce to a new store and replacement agent, then the other can reduce to the

same store and an agent which is (recursively) reactively equivalent to the other replacement.

A related area of research is to develop tools that support reasoning about the agents. For example, it would be useful to be able to expand agent definitions to see the possible traces of user actions which they permit. Another useful tool would allow comparison of the domain and interface models to check that every possible state of the domain is capable of being represented in the interface model. Domain relations are modelled by structures in the domain model which map onto structures in the interface model. However, the constraints on the structures in the interface model effectively provide inference rules concerning the domain model. For example, if a given domain relation is modelled by an interface model structure in which one geom contains another, then the only domain relations that can be expressed in the interface are those which are transitive, asymmetric and irreflexive.

7.4.3 Extensions to the constraint system

The constraints which have been defined are all binary. An interesting possibility would be to develop a constraint system that permitted constraints with other arities. Unary constraints could be used to express the dimensionality of objects. We went some way toward this in section 4.5, when we gave definitions of points and lines in terms of the basic relations. Such unary constraints would restrict the set of binary constraints that can be applied to the object. For example, a point cannot cover a non-point. Higher arity constraints could be used to express relations such as 'inbetween'. At the moment, this kind of constraint can be expressed using a conjunction of the basic relations, but it is not possible to have disjunctions of such constraints. For example, we cannot say that some region either lies between two others, or contains them both.

From the point of view of efficiency of implementation it would be useful to have techniques to find compact ways to represent the constraint graph. In the formulation that we have given every pair of nodes is connected by an arc. In many cases, there will be no restrictions on the labelling between two nodes. In others, the same labelling will apply to every pairing of arcs from two distinct subgraphs. For example, if one *geom* is disjoint from another, then every part of one is disjoint from every part of the other. It would greatly speed up the checking of consistency if this kind of higher-level structuring of the constraint graph could be exploited.

7.4.4 Psychological validity of the basic relations

Abowd [1] proposes a framework, based on work by Norman [113] in which user and system are represented by concurrent agents, each with their own language for describing their possible actions within the system. The system employs a *core* language and the user a *task* language. User-centred design aims to ensure that the core and task languages have a close correspondence: that is, operations which the user requires to complete the task in hand are available within the functional core of the system. However, correspondence between core and task languages is not enough to ensure usability because the user does not have direct access to the underlying system state. Instead, the user and system communicate with each other via an interface. The full model therefore requires an additional two components, input and output, each with their own language, which together constitute the interface. Operations performed by the system are initiated by user actions (articulation) and the resulting change of state of the system is presented to the user as output for observation. The expressiveness of the input and output languages may be considerably constrained relative to the core and task languages. This implies that, whereas a mapping between task and core languages may be relatively straightforward, there is no simple correspondence between tasks and the input actions required to invoke the corresponding core action (the *articulatory* gulf), nor between interface presentation components and the system objects which they represent (the *evaluatory* gulf).

Some work has been done to capture the articulatory and evaluatory links within this model, but this has proved difficult and it is not clear that it can be done successfully. Exclusively task-oriented approaches to describing the articulatory link such as TAG [129] are inadequate to capture fully the frequently exploratory nature of human-computer interaction although they may well suffice at certain levels of description. The evaluatory link, the translation between output and task languages, is a description of how the user interprets the state of the presentation in terms of their model of the underlying system and as such depends on aspects of cognition that are poorly understood.

A user-centred approach to modelling the interface requires us to specify the presentation aspects of the interface in the same terms as the user, thereby narrowing the evaluatory gulf. The notation presented in this thesis goes some way toward doing this, since we describe the picture structure of the interface

using spatial relations which correspond to those used in natural language descriptions of scenes. Of course, we have not explored this point fully and it would be an interesting line of future research to see which spatial relations were employed by users when describing typical graphical user interfaces.

Throughout this thesis we have avoided questions relating to human psychology, cognition and ergonomics: for example, we discussed the use of the notation to describe some typical user-interface styles without considering whether these were examples of good design; we motivated the development of the basic spatial relations by research into logic and topology, rather than perception. From the point of view of good design, it would be interesting to know which spatial relations were to be preferred for the representation of different kinds of information. Another area for research would be to investigate which spatial relations users perceive as having significance in visual representations. This might lead to a richer vocabulary of user interface idioms.

Chapter 8 Conclusions

This chapter is a summary of the research contributions of the thesis.

8.1 Introduction

This chapter is intended to serve as a summary of the research contributions of the thesis. It complements the preceding chapter in which we gave a more detailed account of the research findings and their relation to other research in the field.

8.2 A view of the user interface

A unifying theme of this research has been to demonstrate an integrated approach to the use of constraint programming in the user interface. We started out by proposing that graphical user interfaces can be understood as forms of visual representation. This position gave us a well-motivated framework in which to reason about visual properties of the interface.

The view of the user interface as a form of visual representation has proved to be a useful means to motivate and structure the research. It led us to propose a model for UIMS with the following components

- a constraint store in which the interface state is represented by an integrated system of spatial and structural constraints;
- a sequence of structures which model the instantaneous states of the display;
- constraints between successive states of the display which describe dynamic behaviour of the display.

8.3 Qualitative spatial constraints for UIMS

No previous research into UIMS has considered the use of qualitative spatial constraints. Qualitative constraints allow compact descriptions of the display state of the interface. By contrast, quantitative constraints, such as are used in state-of-the-art UIMS, require much more detailed specifications because they do not specify relations directly in the domain of interest (2.5D space), but instead attempt to use a more general domain ($\mathbb{R}$). When used in the context of local propagation, quantitative constraints cannot handle indeterminate values. This

means that it is impossible to specify correctly spatial relations, such as 'above', 'to the left of', 'within' which arise naturally out of a user-oriented description of the interface.

We gave a formal specification of a system of qualitative spatial constraints based on an axiomatization of 2.5D space, derived from the single relation partCover. The axiomatization is original, albeit based on related work on the formalization of 2D space.

8.4 Use of constraints to describe structure

In order to describe picture structure, it is necessary to integrate the system of spatial constraints with a system of constraints on structure. We have shown how this may be done and given definitions of a set of structural constraints. The use of an integrated system of constraints to manipulate both the structure of objects and relations between them is a novelty in UIMS research. In other constraint-based UIMS, operations on structure are handled separately from the constraint system, typically by means of the class hierarchy in an underlying implementation language.

8.5 Application of the cc() framework to UIMS

We have described a notation for the description of graphical user interfaces and given examples of its use. The notation has a well-founded semantics derived from a computational model (the cc() framework) in which constraints are told atomically to a constraint store.

The notation integrates the system of spatial and structural constraints into a computational framework based on concurrently executing agents communicating through a shared store of constraints. Instantiations of the framework with other constraint systems have been described in the literature: linear equations over real numbers; finite domains; finite trees. Although we do not claim to have made any contribution to the theory of concurrent constraint programming, the use of the cc() framework in the context of UIMS research is new. We have been able to show how to express many idioms of interactive user interfaces naturally and conveniently within the framework.

The advantage of the cc() framework as a vehicle for constraint programming in UIMS is that it provides a simple, but powerful, mechanism for synchronization in the form of the ask operation. This means that it is possible to express process-oriented descriptions of behaviour within the same notation as constraint-oriented descriptions of the display state. Therefore the framework is a

natural choice for an integrated approach to specifying domain, dialogue and presentation components.

The notation allows us to describe aspects of the domain, dialogue and presentation components of the interface without resorting to specialized description languages for each of the components.

8.6 Required labels

We introduced the idea of required labellings. These should not be confused with required constraints, as used in constraint hierarchies. Required labels are needed because we are interested in the set of possible solutions to a given constraint problem, rather than any individual solution. Required labellings allow us to place a lower bound on the set of solutions in the same way that forbidden labellings allow us to place an upper bound on the set.

In formulations of the constraint satisfaction problem which use labelled arcs to represent binary constraints, each constraint is represented as a disjunction of allowed labels, or equivalently, a conjunction of forbidden labels. The constraint may be told provided that there is some solution containing at least one of the allowed labels. In such formulations, the lower limit on the set of solutions is the consistent network in which every arc has exactly one label: this is by definition a single solution.

In conventional formulations of CSP, there is no way to prevent the removal of labels when new constraints are added. For our purposes, it was necessary to allow the incremental addition of information to the network, while having some means to prevent the removal of required labels.

Appendix A Summary of spatial relations

In this appendix we summarize the basic topological and orientation relations defined in Chapter 4. First we give the topological relations which do not distinguish boundary contact. These are illustrated in Table A.4. All the topological relations are derived from a single relation partCover.

Name	Relation	Ref	Sym	Tra
equal	eq(x,y)	*	*	*
disjoint	d(x,y)		*	
coincide	c(x,y)			*
joint	j(x,y)			
hide	h(x,y)			*
upon	u(x,y)			*
iCoincide	$c^{-1}(x,y)$			*
iJoint	$j^{-1}(x,y)$			
iHide	$h^{-1}(x,y)$			*
iUpon	$u^{-1}(x,y)$			*

Table A.1 Basic relations and their inverses.

The relations in Table A.4 are the subcases of joint, distinguished by the extent of the contact between the regions. These distinguish cases of boundary contact. All of these relations have inverses. For the sake of brevity, we do not list the

inverses. None of these relations is symmetric, reflexive or transitive.

Name	Relation
strictJoint	sj(x,y)
weakDisjoint	wd(x,y)
weakUpon	wu(x,y)
weakHide	wh(x,y)
weakCoincide	wc(x,y)
boundary–boundary	bb(x,y)
boundary–region	br(x,y)
region–boundary contact	rb(x,y).

Table A.2 Subcases of joint(x,y). All have inverses: these are not shown.

In Table A.3 we summarize the set of visually distinct relations which were derived in Chapter 4 and define them in terms of the basic relations. The visually distinct relations are disjunctions of the basic relations.

Name	Definition in terms of basic relations
equal(x,y)	eq(x,y)
disjoint(x,y)	d(x,y)
cover(x,y)	h(x,y) $\vee$ c(x,y)
upon(x,y)	u(x,y)
overlap(x,y)	sj(x,y) $\vee$ wh(x,y) $\vee$ wc(x,y)
justOverlap(x,y)	wd(x,y) $\vee$ bb(x,y) $\vee$ br(x,y) $\vee$ rb(x,y)
justUpon(x,y)	wu(x,y)
iCover(x,y)	$h^{-1}(x,y) \vee c^{-1}(x,y)$
iUpon(x,y)	$u^{-1}(x,y)$
iOverlap(x,y)	$sj^{-1}(x,y) \vee wh^{-1}(x,y) \vee wc^{-1}(x,y)$
iJustOverlap(x,y)	$wd^{-1}(x,y) \vee bb^{-1}(x,y) \vee br^{-1}(x,y) \vee rb^{-1}(x,y)$
iJustUpon(x,y)	$wu^{-1}(x,y)$

Table A.3 Visually distinct topological relations defined in terms of the basic relations and their inverses.

The horizontal orientation relations are shown in Table A.4.

Name	Relation	Ref	Sym	Tra
left	l(x,y)			
right	r(x,y)			
weakLeft	wl(x,y)			
weakRight	wr(x,y)			
vAlign	va(x,y)	*	*	

Table A.4 Horizontal orientation relations.

The vertical orientation relations are shown in Table A.4.

Name	Relation	Ref	Sym	Tra
above	a(x,y)			
below	b(x,y)			
weakAbove	wa(x,y)			
weakBelow	wb(x,y)			
hAlign	ha(x,y)	*	*	

Table A.5 Vertical orientation relations.

Horizontal and vertical relations are defined in terms of two partial ordering relations, left and below.

Appendix B Syntax of the notation

In this appendix we give a formal definition of the syntax of the notation, including the abbreviations defined in Chapter 6.

Program :: AgentDefinition [, AgentDefinition]* . Agent

AgentDefinition :: Goal :: Agent | ConstraintGoal == ConstraintExpression

LocalVars :: Name [, Name]* ^

Goal :: Name (Geom [, Geom]*)

ConstraintGoal :: Name (Geom [, Geom]*)

ConstraintExpression :: [LocalVars] Constraint [, Constraint]*

Agent :: [LocalVars] AgentExpression

AgentExpression :: BasicAction | BasicAction -> Agent | Agent
| Agent , Agent | Agent ; Agent | Agent : Agent
DistributedExpression | Goal | stop | **fail**

BasicAction :: (Ask | Tell | Guard) [, (Ask | Tell | Guard)]*

Ask :: ? Constraint

Tell :: ! Constraint

Guard :: [Agent]

Constraint :: SpatialConstraint | GeomConstraint
| GeomConstructor | ConstraintGoal

SpatialConstraint :: [req | not] { Label [, Label]* } (Geom , Geom)

GeomConstraint :: Geom = Geom | Geom << Geom

Geom :: Name | Constructor

Constructor :: Geom @ Geom | Geom + Geom
| Geom - Geom | Geom \ Geom | Geom & Geom
| < Label [: Geom] [, Label [: Geom]]* >
| (Geom [, Geom]*) | { Label [, Label]* }
| Geom . Label | Geom . _ | # Geom

DistributedExpression :: **all** Geom : Geom [Agent]
| **any** Geom : Geom [Agent]
| **collect** Geom : Geom **into** Geom [Agent]

References

[1] Abowd GD (1991) Formal aspects of human-computer interaction. D.Phil. thesis, Oxford University, Programming Research Group. University of York, Dept. of Computer Science, Technical Report YCS (161).

[2] Agha G & Hewitt C (1988) Actors: a conceptual foundation for concurrent object-oriented programming. In *Research directions in object-oriented programming*, ed Shriver B & Wegner P. MIT Press.

[3] Allen JF (1983) Maintaining knowledge about temporal intervals. *Communications of the ACM*, **26**, 11.

[4] Anderson JR (1983) Arguments concerning representations for mental imagery. *Psychological Review*, **85**.

[5] ANSI X3.144 (1988) Computer Graphics - programmer's hierarchical interactive graphical system (PHIGS) functional description. American National Standards Institute.

[6] Ashcroft & Wadge (1977) Lucid, a nonprocedural language with iteration. *Communications of the ACM*, **20**, 7.

[7] Barwise J & Etchemendy J (1991) Visual information and valid reasoning. In Zimmermann W & Cunningham S (eds) *Visualization in mathematics*. Washington, Mathematical Association of America.

[8] Bass L, Little R, Pellegrine R, Reed S, Seacord R, Sheppard S & Szczur MR (1991) The Arch model: Seeheim revisited. *User Interface Developer's Workshop.*

[9] Bass L, Faneuf R, Little R, Mayer N, Pellegrine R, Reed S, Seacord R, Sheppard S & Szczur MR (1992) A metamodel for the runtime architecture of an interactive system. *UIMS Tool Developers Workshop.* SIGCHI Bulletin, 24(1). ACM.

[10] Bass L, Cockton G & Unger C (1993) A reference model for interactive system construction. Report of IFIP working group 2.7 user interface engineering.

[11] van Beek P (1992) Reasoning about qualitative temporal information. *Artificial Intelligence*, **58**, 297-326

[12] Berkeley G (1709) *An essay towards a new theory of vision.*

[13] Berlandier P (1989) PROSE: a constraint language with control structures. *EPIA 89 4th Portuguese conference on AI*, ed Martins JP & Morgado EM. Springer-Verlag.

[14] Bertin J (1983) Semiology of graphics. University of Wisconsin Press. Madison.

[15] Borning AH (1981) The programming language aspects of Thinglab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, **3**, 4.

[16] Borning AH, Duisberg RA, Freeman-Benson B, Kramer A & Woolf M (1987) Constraint hierarchies. *Proceedings of the OOPSLA '87 Conference on Object Oriented Programming Systems Languages and Applications*. ACM.

[17] Buxton W, Fiume E, Hill R, Lee A & Woo C (1983) Continuous hand-gesture driven input. *Proceedings of graphics interface '83.*

[18] Card SK, Mackinlay JD & Robertson GG (1991) A morphological analysis of the design space of input devices. *ACM Transactions on Information Systems*, **9**, 2.

[19] Casner SM (1991) A task-analytic approach to the automated design of graphic pres-

entations. *ACM Transactions on Graphics*, **10**, 2.
[20] Chomsky N (1972) *Syntactic structures.* Mouton & Co. The Hague.
[21] Clarke BL (1981) A calculus of individuals based on connection. *Notre Dame Journal of Formal Logic*, **22**,3.
[22] Clarke BL (1985) Individuals and points. *Notre Dame Journal of Formal Logic*, **26**,1.
[23] Clementini E, di Felici P & van Oosterom P (1993) A small set of formal topological relationships suitable for end-user interaction. In Abel D & Ooi BC (eds) *Advances in spatial databases: Third International Symposium, SSD '93.* LNCS 692. Springer-Verlag.
[24] CMU/SEI (1991) Serpent User's Guide. Software Engineering Institute, Carnegie Mellon University. Report CMU/SEI-91-UG-1.
[25] Cockton G (1987) A new model for separable interactive systems. *Proc. INTERACT '87, 2nd IFIP conference on human-computer interaction.*
[26] Cockton G (1990) Designing abstractions for communication control. In Harrison M & Thimbleby H (eds): *Formal methods in human-computer interaction.* Cambridge University Press.
[27] Colmerauer A (1990) Prolog III. *Communications of the ACM*, **33**, 7
[28] Cooper PR & Swain MJ (1992) Arc consistency: parallelism and domain dependence. *Artificial Intelligence*, **58**. Elsevier.
[29] Coutaz J (1987) PAC, an implementation model for dialog design. *Proc. HCI Interact '87.* North Holland.
[30] Cui Z, Cohn AG & Randell DA (1992) Qualitative simulation based on a logical formalism of space and time. *Proc. AAAI.*
[31] Cui Z, Cohn AG & Randell DA (1993) Qualitative and topological relationships in spatial databases. In Abel D & Ooi BC (eds) *Advances in spatial databases: third international symposium, SSD '93.* LNCS 692. Springer Verlag.
[32] Dance JR, Granor TE, Hill RD, Hudson SE, Meads J, Myers BA & Schulert A (1987) The runtime structure of UIMS supported applications. *Computer Graphics*, **21**, 2.
[33] Dechter R & Pearl J (1988) Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, **34**.
[34] Dix A & Runciman C (1985) Abstract models of interactive systems. In Johnson P & Cook S (eds) *People and Computers: Designing the interface.* Cambridge University Press.
[35] Duce DA, van Liere R & ten Hagen PJW (1990) An approach to hierarchical input devices. *Computer Graphics*, **9**, 1.
[36] Duisberg RA (1987) Animation using temporal constraints: an overview of the Animus system. *Human-Computer Interaction*, **3**
[37] Egenhofer MJ (1989) A formal definition of binary topological relationships. In Litwin W & Schek HJ (eds) *Third int. conf. on foundations of data organization and algorithms.* LNCS 367. Springer-Verlag.
[38] Egenhofer MJ (1991) Reasoning about binary topological relations. In Gunther O & Schek HJ (eds) *Advances in spatial databases: proc. second symposium on large spatial databases.*LNCS 525. Springer-Verlag.
[39] Egenhofer MJ & Sharma J (1993) Topological relations between regions in R^2 and Z^2. In Abel D & Ooi BC (eds) *Advances in spatial databases: third international symposium, SSD '93.* LNCS 692. Springer Verlag.
[40] Faconti GP & Paterno F (1990) An approach to the formal specification of the com-

ponents of an interaction. In Vandoni CE & Duce DA(eds) *Proceedings of Eurographics '90*. North-Holland.
[41] Faconti GP, Zani N & Paterno F (1992) The input model of standard graphics systems revisited by formal specification. In Kilgour A & Kjelldahl H (eds) *Eurographics 92, Computer Graphics Forum,* **11**,3. Blackwell.
[42] de Floriani L, Marzano P & Puppo E (1993) Spatial queries and data models. In Frank AU & Campari I (eds) *Spatial information theory: A theoretical basis for GIS*. LNCS 716. Springer-Verlag.
[43] Foley JD, Kim WC, Kovacevic S & Murray K (1991) UIDE - An intelligent user interface design environment. In Sullivan J and Tyler S (eds) *Architectures for intelligent interfaces:Elements and prototypes*. Addison Wesley.
[44] Foley JD & Wallace VL (1974) The art of natural graphic man-machine conversation. *Proc. IEEE*, **62**,4.
[45] Foley JD & van Dam A (1982) Fundamentals of interactive computer graphics. Addison Wesley.
[46] Freeman-Benson B (1989) A module mechanism for constraints in Smalltalk. *OOPSLA 1989 Proceedings*. ACM publications.
[47] Freeman-Benson B (1990) Kaleidoscope: mixing objects, constraints, and imperative programming. *ECOOP/OOPSLA Proceedings*.
[48] Freeman-Benson B, Maloney J & Borning A (1990) An incremental constraint solver. *Communications of the ACM*, **33**, 1
[49] Freksa C (1992) Temporal reasoning based on semi-intervals. *Artificial Intelligence*, **54**
[50] Freksa C (1992) Using orientation information for qualitative spatial reasoning. In Frank AU, Campari I & Formentini U (eds). *Theories and methods of spatio-temporal reasoning in geographic space*. LNCS 639. Springer-Verlag.
[51] Freuder EC (1978) Synthesizing constraint expressions. *Comm. ACM*, **21**, 11.
[52] Freuder EC (1982) A sufficient condition for backtrack-free search. *J. ACM*, **29**, 1.
[53] Freuder EC (1985) A sufficient condition for backtrack-bounded search. *J. ACM*, **32**,4.
[54] Fruhwirth T (1992) Constraint simplification rules. Technical report ECRC-92-18. ECRC, Munchen, Germany.
[55] Galton F (1883) Inquiries into human faculty and its development. Macmillan.
[56] Giuse DA (1990) Efficient knowledge representation systems. *Knowledge Engineering Review*, **5**.
[57] Gould JD, Ukelson J & Boies SJ (1993) Improving application development productivity by using ITS. *Int. J. Man–Machine Studies*, **39**.
[58] Gray PD, Waite KW & Draper SW (1990) Do-it-yourself iconic displays: reconfigurable iconic representations of application objects. In Diaper D (ed) *Human–computer interaction - INTERACT '90*. Elsevier. North-Holland.
[59] Gray PD, England D & McGowan S (1993) XUAN: enhancing the UAN to capture temporal relationships between actions. *HCI '93*.
[60] Green M (1985) Report on dialogue specification tools. In Gunther Pfaff (ed) *Proceedings of the workshop on user interface management systems*, Seeheim November 1983. Springer-Verlag.
[61] Green TRG(1989) Cognitive dimensions of notations. In Sutcliffe A & Macaulay L

(eds) *People and Computers V.* Cambridge University Press.
[62] Green TRG (1991) Describing information artefacts with cognitive dimensions and structure maps. In Diaper D & Hammond (eds) *Proceedings. of HCI '91: Usability Now.*
[63] Han CC & Lee CH (1988) Comments on Mohr & Henderson's path consistency algorithm. *Artificial Intelligence*, **36**.
[64] Harel D (1988) On visual formalisms. *Communications of the ACM.*
[65] Haridi S *et al* (1995) Concurrent constraint programming at SICS with the Andorra Kernel Language.
[66] Harrison M & Dix A (1990) A state model of direct manipulation in interactive systems. In Harrison M & Thimbleby H (eds): *Formal methods in human-computer interaction.* Cambridge University Press.
[67] Hartson HR & Gray PD (1992) Temporal aspects of tasks in the user action notation. *Human-Computer Interaction*, **7**.
[68] Hartson HR & Hix D (1989) Human-computer interface development: concepts and systems for its management. *ACM Computing Surveys*, **21**, 1.
[69] Hartson HR, Siochi AC & Hix D (1990) The UAN: a user-oriented representation for direct-manipulation interface designs. *ACM Transactions on Information Systems*, **8**.
[70] van Hentenryck P, Simonis H & Dincbas M (1992) Constraint satisfaction using constraint logic programming. *Artificial Intelligence* **58**. Elsevier.
[71] Hernandez D (1993) Maintaining qualitative spatial knowledge. In Frank A & Campari I (eds) *Spatial information theory: a theoretical basis for GIS.* LNCS 716. Springer-Verlag.
[72] Hernandez D & Zimmermann W (1993) Default reasoning and the qualitative representation of spatial knowledge. In Habel et al (eds). *Defaults and prototypes – Non-monotonic reasoning for language and knowledge processing.* Springer-Verlag.
[73] Hill RD (1986) Supporting concurrency, communication and synchronization in human-computer interaction - the Sassafras UIMS. *ACM Trans. on Graphics.* 5(3). ACM
[74] Hill RD & Herrmann M (1989) The structure of Tube - A tool for implementing advanced user interfaces. In *Proc. Eurographics '89.*
[75] Hill RD (1993) The Rendezvous constraint maintenance system. *Proc UIST '93.* ACM.
[76] Hoare CAR (1985) Communicating sequential processes. Prentice Hall International.
[77] Hoover R (1987) Incremental graph evaluation. PhD Thesis. Deptartment of Computer Science, Cornell University, Ithaca, NY.
[78] Hyvonen E (1992) Constraint reasoning based on interfval arithmetic: the tolerance propagation approach. *Artificial Intelligence*, **58**. Elsevier.
[79] ISO/IS 9636 (1991) Information processing systems, Computer Graphics, Interface techniques for dialogues with graphical devices.
[80] Jackendoff R (1983) Semantics and cognition. MIT Press.
[81] Jaffar J & Lassez J (1987) Constraint logic programming. ACM
[82] Jaffar J, Michaylov S, Stuckey PJ, Yap RHC (1992) The CLP(R) language and system. *ACM Transactions on Programming Languages and Systems.* ACM Press.
[83] Kahn KM & Saraswat VA (1990) Actors as a special case of concurrent constraint programming. *ECOOP/OOPSLA Proceedings.*
[84] Kahn KM, Tribble ED, Miller MS & Bobrow DG (1987) Vulcan: logical concurrent objects. In Shriver B & Wegner P (eds) *Research directions in object-oriented program-*

ming. MIT Press series in computer systems.
[85] Kamada T & Kawai S (1991) A general framework for visualizing abstract objects and relations. *ACM Transactions on Graphics*, **10**, 1.
[86] Kant I (1781) Kritik der reinen Vernunft [Critique of Pure Reason]. Translated by Kemp Smith N. Macmillan. 1929.
[87] de Kleer J & Sussman GJ (1980) Propagation of constraints applied to circuit synthesis. Circuit Theory and Applications, **8**.
[88] Kovacevic, S (1992) A compositional model of human-computer interaction. PhD thesis. Dept. of Electrical Engineering and Computer Science. George Washington University. Washington DC.
[89] Kramer GA (1992) A geometric constraint engine. *Artificial Intelligence*, **58**, Elsevier.
[90] Krasner GE &Pope ST (1988) A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *J. Object Oriented Prog.*
[91] Kurtenbach G & Buxton W (1991) Issues in combining marking and direct manipulation techniques. *UIST '91*. ACM.
[92] Kurtenbach G & Buxton W (1991) GEdit: a test-bed for editing by continuous gestures. *SIGCHI Bulletin.*
[93] Larkin JH & Simon HA (1987) Why a diagram is (sometimes) worth a thousand words. Cognitive Science, **11**.
[94] Leler W (1988) Constraint programming languages: their specification and generation. Addison Wesley.
[95] Levesque HJ (1986) Making believers out of computers, *Artificial Intelligence*, **30**.
[96] Levesque HJ (1986) Logic and the complexity of reasoning. *Journal of Philosophical Logic*, **17**, 355-389.
[97] Mackinlay JD (1986) Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics*, **5**, 2.
[98] Mackinlay JD, Card SK & Robertson GG (1990) A semantic analysis of the design space of input devices.
[99] MacKinnon D, McCrum W & Sheppard D 1990. *An introduction to open systems interconnection.* Computer Science Press. Freeman & Company. New York.
[100] Mackworth AK (1977) Consistency in networks of relations. *Artificial Intelligence*, **8**.
[101] Mackworth AK (1992) The logic of constraint satisfaction. *Artificial Intelligence*, **58**, Elsevier.
[102] McMorran MA & Nicholls JE (1989) *Z user manual.* IBM Technical Report TR12.274.
[103] Milner R (1980) *A calculus of communicating systems*. LNCS. Springer Verlag.
[104] Minsky M (1984) Manipulating simulated objects with real-world gestures using a force and position sensitive screen. *Computer Graphics*, **18**, 3.
[105] Mohr R & Henderson TC (1986) Arc and path consistency revisited. *Artificial Intelligence*, **28**.
[106] Montanari U (1974) Networks of constraints: fundamental properties and applications to picture processing. *Inform. Sci.* **7**.
[107] Mukerjee A & Joe G (1990) A qualitative model for space. *Proc. 8th National Conf.*

on AI. AAAI. MIT Press.
[108] Myers BA (1990) A new model for handling input. *ACM Transactions on Information Systems,* **8**, 3.
[109] Myers BA, Giuse DA, Dannenberg RB, Vander Zanden B, Kosbie DS, Marchal P, Mickish A and Pervin E (1990) Comprehensive support for graphical, highly interactive user interfaces: The Garnet user interface development environment. *IEEE Computer,* **23**, 11
[110] Myers BA, Giuse DA, Dannenberg RB, Vander Zanden B, Kosbie DS, Marchal P, Mickish A, Pervin E and Kolojejchik JA (1990) The Garnet toolkit reference manuals: Support for highly interactive graphical user interfaces in Lisp. Tech report CMU-CS-90-117, Computer Science Dept, Carnegie Mellon University.
[111] Neches R, Foley J, Szekely P, Sukaviriya P, Luo P, Kovacevic S & Hudson S (1993) Knowledgeable development environments using shared design models. *Proceedings of the intelligent interfaces workshop*, Orlando.
[112] Newman W (1968) A system for interactive graphical programming
[113] Norman DA (1988) *The psychology of everyday things.* Basic Books.
[114] Patterson JF, Hill RD, Rohall SL & Meeks WS (1990) Rendezvous: an architecture for synchronous multi-user applications. *CSCW '90 Proceedings.* ACM Publications.
[115] Pineda LA (1989) GRAFLOG: a theory of semantics for graphics with applications to human–computer interaction and CAD systems. PhD thesis, University of Edinburgh.
[116] Pineda LA (1992) Reference, synthesis and constraint satisfaction. In Kilgour A & Kjelldahl L (eds) *Eurographics '92. Computer Graphics Forum*, **11**, 3.
[117] Pineda LA, Klein EH & Lee J (1988) GRAFLOG: understanding graphics through natural language. *Computer Graphics Forum*, **7**.
[118] Pratt I
[119] Pylyshyn ZW (1973) What the mind's eye tells the minds brain: A critique of mental imagery. *Psychological Bulletin,* **80**.
[120] Rosenthal DSH, Michener JC, Pfaff G, Kessner R & Sabin M (1982) The detailed semantics of graphics input devices. *Computer Graphics,* **16**, 3.
[121] Rubine DH (1990) The automatic recognition of gestures. Doctoral thesis. Dept of Computer Science, Carnegie Mellon University.
[122] Runciman C (1990) From abstract models to functional prototypes. In Harrison M & Thimbleby H (eds): *Formal methods in human-computer interaction.* Cambridge University Press.
[123] Sannella, M (1995) The skyblue constraint solver and its applications. In press.
[124] Saraswat VA (1987) The concurrent logic programming language CP: definition and operational semantics. Proceedings of the SIGACT-SIGPLAN symposium on principles of programming languages. ACM
[125] Saraswat VA (1990) *Concurrent constraint programming.* ACM.
[126] Saraswat VA (1993) *Concurrent constraint programming.* MIT Press.
[127] Saraswat VA, Kahn K & Levy J (1990) Janus: A step toward distributed constraint programming. *Proc. North American Conference on Logic programming.*
[128] Schapiro E (1987) Concurrent prolog. MIT.
[129] Schiele F & Green T (1990) HCI formalisms and cognitive psychology: the case of task-action grammar. In Harrison M & Thimbleby H (eds): *Formal methods in human-*

computer interaction. Cambridge University Press.
[130] Schlichtmann H (1985) Characteristics of the semiotic system 'Map Symbolism'. *Cartographic Journal,* **22**.
[131] Shneiderman B (1982) The future of interactive systems and the emergence of direct manipulation, *Behaviour and Information Technology,* **1**, 3.
[132] Simons PM (1982) Three essays in formal ontology. In Smith B (ed) *Parts and moments, studies in logic and formal ontology.* 1982 Philosophia Verlag.
[133] Soede D, Arbab F, Herman I & ten Hagen PJW (1991) The GKS input model in MANIFOLD. *Computer Graphics Forum,* **10**, 3.
[134] Sowa JF (1991) Principles of semantic networks: Explorations in the representation of knowledge. Morgan Kaufmann.
[135] Spivey JM (1988) *Understanding Z: A specification language and its formal semantics.* Cambridge Unviersity Press.
[136] Spivey JM (1989) *The Z notation: A reference manual.* Prentice-Hall.
[137] Steele GL (1980) The definition and implementation of a computer programming language based on constraints. PhD thesis. MIT
[138] Stenning & Oberlander J (1993) A cognitive theory of graphical and linguistic reasoning: logic and implementation.
[139] Sufrin B (1982) Formal specification of a display-oriented text editor. *Science of Computer Programming,* **1**. North Holland.
[140] Sufrin B & He J (1990) Specification, analysis and refinement of interactive processes. In Harrison M & Thimbleby H (eds): *Formal methods in human-computer interaction.* Cambridge University Press.
[141] Sukaviriya P, Foley J & Griffith T (1993) A second generation user interface design environment: The model and the runtime architecture. *Proc. INTERCHI 1993.* ACM Publications, New York.
[142] Sussman G & Steele GL (1980) CONSTRAINTS – a language for expressing almost-hierarchical descriptions. *Artificial Intelligence Journal,* **14**.
[143] Sutherland IE (1963) Sketchpad: A man-machine graphical communication system. AFIPS Spring Joint Computer Conference.
[144] Szekely P (1990) Template-based mapping of application data to interactive displays. *Proc. UIST '90.* ACM.
[145] Szekely P, Luo P & Neches R (1992) Facilitating the exploration of interface design alternatives: The HUMANOID model of interactive design. *Proc. SIGCHI '92.*
[146] Szekely P, Luo P & Neches R (1993) Beyond interface builders: model-based interface tools. *Proc. INTERCHI '93.* ACM.
[147] Takahashi S, Matsuoka S & Yonezawa A (1991) A general framework for bi-directional translation between abstract and pictorial data. *UIST '91.*
[148] Took RK (1990) Surface interaction: A paradigm and model for separating application and interface. In *Proc CHI '90.* ACM.
[149] Took RK (1991). The active medium: a conceptual and practical architecture for direct manipulation. In *Proceedings of HCI '91: People and Computers VI.* Cambridge University Press.
[150] Tufte ER (1983). *The visual display of quantitative information.* Cheshire, Con-

necticut. Graphics Press.
[151] Tufte ER (1990).*Envisioning information.* Cheshire, Connecticut. Graphics Press.
[152] Vieu L (1993) A logical framework for reasoning about space. In Frank AU & Campari I (eds) *Spatial information theory: A theoretical basis for GIS.* LNCS 716. Springer-Verlag.
[153] Vilain M & Kautz HA (1986) Constraint propagation algprithms for temporal reasoning. In *Proceedings AAAI-86.*
[154] Wadler P (1987) Views: a way for pattern matching to cohabit with data abstraction. *Proceedings 14th ACM Principles of Programming Languages Conference.*
[155] Wagner G (1994) *Vivid Logic.* LNCS 764. Springer-Verlag.
[156] Wegner P (1987) The object-oriented classification paradigm. In Shriver B & Wegner P (eds) *Research directions in object-oriented programming.* MIT.
[157] Wiecha C, Bennet W, Boies S, Gould J & Greene S (1990) ITS: a tool for rapidly developing interactive applications. *ACM Transactions on Information Systems,* **8**, 3.
[158] Wilk MR (1991) Equate: an object-oriented constraint solver. *OOPSLA.*
[159] Wolf CG, Rhyne JR & Ellozy HA (1989) The paper-like interface. In Salvendy G & Smith MJ (eds) *Designing and using human–computer interfaces and knowledge-based systems.* Elsevier, Amsterdam.
[160] Wood CA and Gray PD (1992) User interface-application communication in the Chimera user interface management system. *Software-Practice and Experience*, **22**(1). Wiley & Sons Ltd.
[161] Woods WA (1991) Understanding subsumption and taxonomy: A framework for progress. In Sowa JF (ed) *Principles of semantic networks: Explorations in the representation of knowledge.* Morgan–Kauffmann.
[162] Wyk J van (1982) A high-level language for specifying pictures. *ACM Trans. Graph.* **1**(2)
[163] Zadrozny W (1991) Logical dimensions of some graph formalisms. In Sowa JF (ed) *Principles of semantic networks: Explorations in the representation of knowledge.* Morgan–Kauffmann.
[164] van der Zanden B, Myers BA, Szekely P & Giuse D (1991) The importance of pointer variables in constraint models. *UIST 1991.*